

# problemspace

An introduction to the art and science of solving problems with computation.

(and some Python)



*Palmer*

This documentation is released under a Creative Commons Attribution-ShareAlike 4.0 International License.

<http://creativecommons.org/licenses/by-sa/4.0/>

Attribution must include the following four lines:

Copyright © 2014 James Dean Palmer and others.

All rights reserved.

[problemspace.org](http://problemspace.org)

Problemspace™ is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

# Contents

Preface	i
1 Problem Solving	1
1.1 <i>The Origins of Computation</i> . . . . .	2
1.2 <i>Describing Computation with Language</i> . . . . .	6
1.3 <i>A Problem Solving Strategy</i> . . . . .	9
1.4 <i>How to think like a Computer Scientist</i> . . . . .	16
2 Getting Started with Python	29
2.1 <i>Why Python?</i> . . . . .	30
2.2 <i>Installing Python</i> . . . . .	32
2.3 <i>Running Python</i> . . . . .	35
2.4 <i>Programs</i> . . . . .	38
2.5 <i>Comments</i> . . . . .	39
3 Primitive Expressions	49
3.1 <i>Variables</i> . . . . .	50
3.2 <i>Booleans</i> . . . . .	55
3.3 <i>Numbers</i> . . . . .	58
3.4 <i>Strings</i> . . . . .	67
3.5 <i>Lists and Dictionaries</i> . . . . .	74
3.6 <i>None</i> . . . . .	75
3.7 <i>Comparing Types</i> . . . . .	76
4 Compound Expressions	87

4.1	<i>Function Composition</i>	88
4.2	<i>Method Chaining</i>	90
4.3	<i>Operator Precedence</i>	91
4.4	<i>Short-Circuit Evaluation</i>	93
4.5	<i>Truthiness</i>	95
4.6	<i>Augmented Assignment Operators</i>	96
4.7	<i>String Formatting</i>	98
5	Functions	109
5.1	<i>Defining Functions</i>	110
5.2	<i>Parameters</i>	110
5.3	<i>Return Values</i>	114
5.4	<i>Docstrings</i>	115
5.5	<i>Scope</i>	116
5.6	<i>Code Repetition</i>	119
5.7	<i>Top-Down Design</i>	121
5.8	<i>Recursion *</i>	123
5.9	<i>Functions as Values *</i>	124
6	Collections	139
6.1	<i>Lists</i>	139
6.2	<i>Dictionaries</i>	145
6.3	<i>Tuples</i>	147
6.4	<i>Sets</i>	149
7	Iteration	161
7.1	<i>for Loops</i>	161
7.2	<i>while Loops</i>	168
7.3	<i>break and continue</i>	170
7.4	<i>Pattern Generalization</i>	171
8	Input and Output	179
8.1	<i>Files</i>	180

8.2	<i>Serialization</i>	181
8.3	<i>JSON *</i>	183
9	<b>Classes and Objects</b>	191
9.1	<i>Classes</i>	192
9.2	<i>Methods</i>	193
9.3	<i>Fields</i>	194
9.4	<i>Encapsulation</i>	195
9.5	<i>Inheritance</i>	197
9.6	<i>Polymorphism</i>	199
9.7	<i>Comparing Objects</i>	199
9.8	<i>Error Handling</i>	200
9.9	<i>Class Members *</i>	203
9.10	<i>Properties *</i>	204
10	<b>Testing</b>	211
10.1	<i>Fail Early</i>	213
10.2	<i>Contracts</i>	214
10.3	<i>Logging</i>	216
10.4	<i>Unit Testing</i>	220
10.5	<i>Equivalence Classes</i>	222
10.6	<i>Regression Testing</i>	223
10.7	<i>Test Driven Development</i>	224
11	<b>Regular Expressions *</b>	229
11.1	<i>Concatenations</i>	230
11.2	<i>Alternations</i>	231
11.3	<i>Repetition</i>	232
11.4		232
A	<b>Pair Programming</b>	235
B	<b>Turtle Graphics</b>	237
C	<b>Unicode</b>	239

D	Syntax Diagrams	245
	List of Strategies	257
	List of Principles	258
	List of Tables	259
	Index	261

# Preface

*Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration.*

*Stan Kelly-Bootle*

If you are looking for an excellent Python reference, this isn't it. This is a textbook about **problem solving** that incidentally uses Python. In other words, this is a book about computer science.

## *Problem Solving Driven*

There are a good many texts you might choose as an introduction to computer science and some of them even have the words “problem solving” in the title but very few of these texts discuss problem solving processes and techniques in a structured way. Instead, they make the assumption that by giving you a few simple problems you will, perhaps by osmosis, learn problem solving.

The approach of this text is far more direct. In Chapter 1 we will introduce a problem solving process and as the book progresses we will discuss how programming languages can be used as a tool for representing and solving problems. We will practice these problem solving strategies and skills with examples. **Lots** of examples.

In addition to the main problem solving process introduced in Chapter 1, we will also describe problem solving heuristics and programming practice principles. These are emphasized in yellow boxes and collected in a list of Strategies and Principles at the end of the text.

## *Accessibility*

This text was designed for beginning students. We assume that students have had high school algebra and geometry but not much else. We don't assume that students have had any programming experience.

We also don't make assumptions about how students want to read this text.

This text is available as a PDF, which as of this writing, is less than three megabytes. It's designed to look good on desktops, laptops, and tablets. It's also beautiful when printed on paper.

### *Scaffolding*

This text has been designed so that at the end of each chapter students know a useful subset of Python that builds on their knowledge from previous chapters. Many of the problems are also scaffolded such that Problem Z builds on Problem Y, which in turn builds on Problem X. In this way students get to focus on different practices and design decisions associated with a single problem.

### *Active Learning Ready*

For students to learn they must read, write, collaborate, and be engaged in solving problems. And while this text can't start a class discussion or launch a group design activity, we provide ideas and resources for adding active learning elements to the classroom. If you are an instructor and would like the Problemspace Instructor's Manual, please visit <http://problemspace.org> for more information.

### *Curated Examples*

The code presented in this text has been carefully curated to represent good coding style, best-practice design, and realistic (if not common) computer science problems. Code is syntax highlighted to emphasize which entities have different meanings.

### *Rich Reading Tools*

If you are reading this on an electronic device, this is not a passive textbook. We provide a comprehensive glossary of important computer science terms for each chapter, a robust bibliography of important papers and further reading for each chapter, and we pervasively hyperlink internal and external resources for further reading or exploration.

### *Practice Resources*

I can't overemphasize the importance of practice. Student success in computer science is critically linked to the quality and quantity of practice that students engage in. That's why we have done our best to provide lots of problems for



students to work. There are over 200 reading questions with twenty questions for each chapter, over 200 carefully scaffolded exercises with at least twenty problems for each chapter, and worksheets (of smaller problems) and projects (of larger problems) available separately.

### *Instructor Resources*

Resources available for instructors include over 100 slides in PowerPoint and PDF format in both 4:3 and 16:9 aspect ratios, an instructor's manual available (only) to instructors, learning management friendly solutions to reading questions, and solutions to selected exercises in the text.

### *Freedom*

This text is also free in two different senses. Students can download it for free but it's also released under a permissive Creative Common license which allows you to remix it and adapt it for your classes. If you do, we just ask that you credit the original work appropriately.

## **Contributing**

Corrections and contributions to this effort are appreciated! Please help us create the best CS1 text possible. You can submit corrections, changes, and additions to our repository, mailing list or by emailing Dr. Palmer directly.

- Website: <http://problemspace.org>
- Git repository: <https://github.com/jdpalmer/problemspace>
- Discussion: <https://groups.google.com/d/forum/problemspace>
- Email: James.Palmer at nau dot edu

## **Conventions Used in This Book**

The following font conventions are used in this book:

- **Bold** is used for:
  1. New terms where they are first used or defined.
  2. Words intended to be emphasized.
- `Constant Width` is used for:

*x*

1. Interactions with the console or command line.
2. Program listings.
3. Names, keywords, and literals used in program listings.
4. Domain names and URLs.

Reading questions appear at the end of each chapter and are labeled with the chapter number, a period, and then the question number. Thus, **3.17** refers to the 17<sup>th</sup> reading question in Chapter 3.

Exercises also appear at the end of each chapter and are labeled with the chapter number, a dash, and then the exercise number. Thus, **1-13** refers to the 13<sup>th</sup> exercise in Chapter 1.

Program listings contain colored syntax highlighting with line numbers typeset to the left of the code (in Python, line numbers are not part of the code listing itself) and strings of text use subscripted half squares (□) to represent spaces:

```
1 print("Hello.")
2 print("This□is□going□to□be□fun!")
```

The right arrow (→) is used within code listings to represent the result of an expression. While the expression to the left of the arrow is valid Python code the arrow itself is not. The right arrow corresponds to what you would see if the expression was evaluated interactively in Python (more or less). Thus, while we might typeset simple arithmetic statements in a program listing like this:

```
1 a = 1
2 b = 2
3 a + b → 3
4 a → 1
```

The same code as you might type see it in an interactive Python session would be displayed like this:

```
1 >>> a = 1
2 >>> b = 2
3 >>> a + b
4 3
5 >>> a
6 1
```

When a program listing includes the word “Output” followed by a colon, the remainder of the listing represents the output (textual or graphical) from the

program. Program output is not syntax highlighted, nor does it include line numbers:

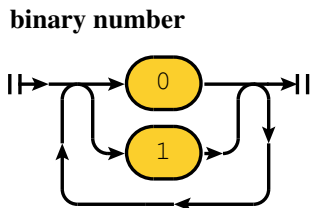
```
1 a = 1
2 b = 2
3 print(a + b)
```

```
1 Output:
2 3
```

The result of an expression and output are two very different things as we will discuss later in the text.

Throughout this text we will describe Python’s grammar in a semi-formal way using railroad diagrams (also called syntax diagrams), which provide a graphical representation of the grammar.

A binary number, for example, consists of any combination of 0s and 1s, which we can draw as:



Railroad diagrams can be interpreted using these rules<sup>1</sup>:

1. Start with the vertical bar(s) on the left and follow the tracks in the direction specified by the arrows.
2. You will encounter keywords, symbols, and literals in rounded boxes (called terminals).
3. You will encounter rules in rectangles (called non-terminals).
4. If a diagram begins with a single bar then any amount of whitespace (e.g., spaces, tabs) is permitted between terminals and non-terminals.
5. If a diagram begins with a double bar then whitespace is not permitted between terminals and non-terminals.

---

1. Most of the conventions used in our railroad diagrams were borrowed from Niklaus Worth’s “The Programming Language Pascal” [24]. The use of bars and double bars was influenced by Douglas Crawford’s “JavaScript: The Good Parts” [7].

6. Any path that can be made to terminate at the vertical bar(s) on the right by following the tracks is a legal syntactic sequence.
7. Any syntactic sequence that can not be made by following the tracks is illegal.

Please note that in many cases the grammar we present has been simplified significantly both for clarity and to emphasize the elements of Python discussed in this book.

Hyperlinks are used pervasively in this book. References to chapters, figures, tables, keywords, and outside resources almost always include hyperlinks. Hyperlinks that link to anchors within this text are colored red. Hyperlinks that link to external websites or resources are colored magenta.

### How This Book is Organized

Each chapter in this book is about problem solving. The first chapter presents a problem solving process, while subsequent chapters discuss how computational elements like iteration, functions, and abstraction can be used to model problems and answer questions. Problem solving heuristics and practice is emphasized over complete descriptions of each nook and cranny of the Python language and its libraries.

Chapter 1, **Problem Solving**, describes the origin of computational machines and introduces the four step problem solving process used throughout this text.

Chapter 2, **Getting Started with Python**, describes where Python is used in the wild, why it is being used in this book, and how to install it on a personal computer.

Chapter 3, **Primitive Expressions**, describes how values are represented, stored, and manipulated with simple expressions.

Chapter 4, **Compound Expressions**, describes how simple expressions can be combined into more complicated expressions.

Chapter 5, **Functions**, describes how expressions and statements can be collected into new functions.

Chapter 6, **Collections**, describes simple yet powerful data structures which can be used to represent sequences and relationships in a problem.

Chapter 7, **Iteration**, introduces powerful constructs for repeating calculations.

Chapter 8, **Input and Output**, describes how Python interacts with files and file based structures for transporting data.

Chapter 9, **Classes and Objects**, describes Object-Oriented programming as a mechanism for organizing code and creating new “kinds” of data with type specific state and behavior.

Chapter 10, **Testing**, describes why testing is an integral part of implementation and how testing can create safer, better designed programs.

Chapter 11, **Regular Expressions**, describes a language within a language for breaking up text into meaningful components.



# 1

## Problem Solving

*Computer science is no more about computers than astronomy is about telescopes.*

*Edsger Dijkstra*

```
1 overview = ["The_Origins_of_Computation",
2             "Describing_Computation_with_Language",
3             "A_Problem_Solving_Strategy",
4             ["Step1:_Analyze_the_problem",
5             "Step2:_Plan_Your_Solution",
6             "Step3:_Implement_and_Test_Your_Solution",
7             "Step4:_Refactor_Your_Solution"],
8             "How_to_Think_Like_a_Computer_Scientist",
9             "Glossary",
10            "Reading_Questions",
11            "Exercises",
12            "Chapter_Notes"]
```

This book is about two things: learning to write software and solving problems. These two goals are inexorably linked. Understanding programming language syntax is useless if you can't use this as a tool to solve problems and understanding how to solve computational problems isn't useful if you can't realize this in working designs. This text takes a balanced and pragmatic approach to computational problems solving by introducing the bits of language that we think are most important in solving complex problems and then concentrating on developing computational thinking[23] and problem solving skills.

Along the way we'll emphasize approaches for breaking up problems, abstracting concepts, communicating ideas and developing best practices and aesthetic



Figure 11: A Pascaline calculator adds numbers using dials, toothed wheels, and falling weights. While incredibly clever, the complicated internal carry mechanism made it unreliable.

designs. We hope you are ready for an exciting journey where Python will help us create art, make music and solve problems.

At its core computer science is about teamwork, engineering, creativity, and problem solving. It's a discipline that gives you the background crucial to explore careers in digital forensics, bioinformatics, computational medicine, industrial process control, robotics, game design, web development, or network administration. From science and engineering to health care and the environment, computer scientists make a difference.

## 1.1 The Origins of Computation

Calculation tools such as the abacus, straightedge, and compass date back to antiquity but the word **computer** first appeared in the 1600s and for a long time it referred to an occupation. A computer was originally a person who performed mathematical computations for a living.

One of the first machines that performed mathematical calculations was the Pascaline calculator. The calculator's genesis was rooted in the appointment of Étienne Pascal as the king's commissioner of taxes to Rouen, France in 1639. Rouen was at the epicenter of the French Wars of Religion and at the time of Étienne's appointment the city's records were in chaos. Étienne spent countless weary hours calculating and re-calculating taxes owed and paid. His son, Blaise Pascal, imagined that a machine rather than a man could be made to do these simple calculations and in 1642, at the age of 18, he designed and built the first mechanical calculator [5, 11]. The Pascaline calculator, featured in Figure 11, was limited to addition and subtraction, although multiplication could be accomplished with manually repeated additions.



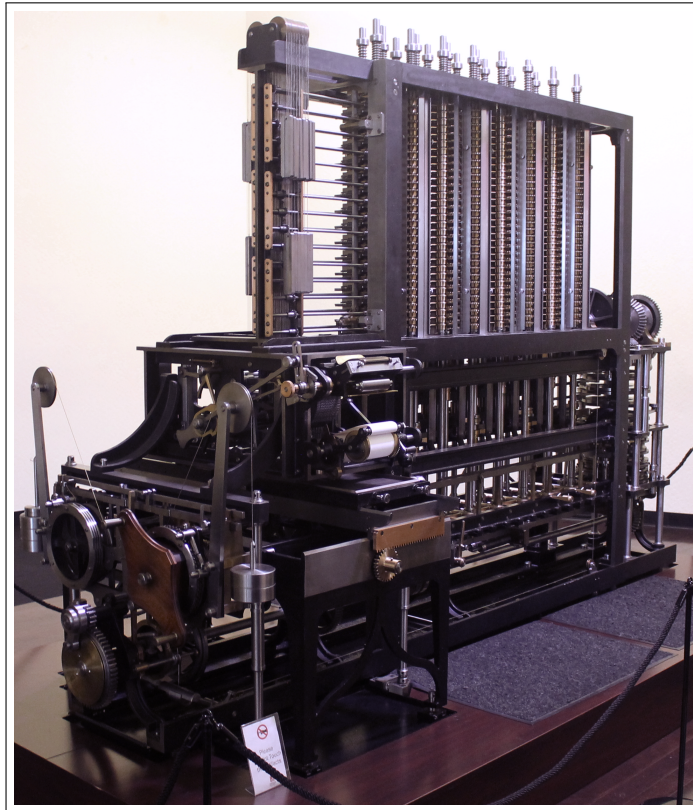


Figure 12: *In 1991, the London Science Museum completed the construction of Difference Engine No. 2 using Charles Babbage's original plans from the 1830s.*

One of the most revolutionary computational tools of the seventeenth and eighteenth centuries was the printing press. As the cost and speed of printing improved, tables and catalogs of common calculations became essential to bankers, architects, navigators, and engineers. Printed tables were produced by computers (of the human variety) who calculated and checked equations, and then manually typeset the numerical results.

In 1821, Charles Babbage and his friend and noted astronomer, John Herschel, were inspecting a volume of astronomical tables which Herschel had commissioned. Much to their dismay, the tables were full of errors and the process of finding the errors was tedious and exhausting. Errors were common in the printing process and often resulted in lengthy publications of errata. After finding error after error Babbage finally exclaimed, "I wish to God these



Figure 13: *Lady Augusta Ada Lovelace described the first program designed to be run by a machine.*

calculations had been executed by steam!” [13]

In 1823, Babbage convinced the British government to fund the construction of Difference Engine No. 1, a machine that could evaluate polynomial expressions. Unlike the Pascaline calculator, Babbage’s machine used more sophisticated machinery and was designed in such a way that the machine would seize rather than give an incorrect calculation. Furthermore, the Difference Engine would automatically typeset the results removing human error even from the typesetting process. While one small scale demonstration unit called “the beautiful fragment” was produced in 1832, the project stalled a year later due to the substantial amount of time and money needed to complete it.

This did not detour Babbage who continued developing his ideas in an improved Difference Engine No. 2 (see Figure 12) and an even more ambitious project he called the Analytical Engine. This new machine would be capable of more general computations (not just polynomial evaluations like his difference engines) using loops, conditional branching, and an integrated memory. In work published by one of Babbage’s collaborators, Lady Augusta Ada Lovelace, Lovelace described the steps the Analytical Engine must perform to calculate

Bernoulli numbers - essentially writing the first program for a machine [9]. Lovelace also discussed how computational engines could be used to manipulate quantities that weren't mathematical such as letters of the alphabet and notes of music - a vision that was decades ahead of its time.

While Babbage was unable to complete the construction of his analytical engines during his life, others would soon have more success.

One inventor, Herman Hollerith, worked for the U.S. Census Office during the 1880 census and recognized that machines could replace people in tabulating results. The 1880 census would take the U.S. Census Office eight years to complete leading some at the Census Office to worry they would soon have to count two sets of census data at the same time. Hollerith developed practical electro-mechanical computers that used punch cards to store both data and programs and won a contract to tabulate the 1890 census. With electro-mechanical computation, Hollerith's company tabulated the data for the 1890 census in just six weeks and the entire 1890 census was completed in under three years.

In 1896 Hollerith would found the Tabulating Machine Company, which would eventually be renamed International Business Machines (IBM).

In 1946, ENIAC (short for Electronic Numerical Integrator And Computer) became the first electronic computer [17, 11]. This was an important milestone in computing history as the ENIAC was roughly a thousand times faster than electro-mechanical machines of the time. The ENIAC contained 17,468 vacuum tubes (electronic switches used to implement logical operations) and over 70,000 resistors. At a weight of 30 tons it occupied 680 square feet of space.

By the 1960s innovations in transistors (which replaced vacuum tubes as smaller and cheaper alternatives for representing digital logic) and integrated circuitry ignited a movement of exponential improvement in computer hardware [21]. Integrated circuits were becoming smaller, faster, and cheaper.

Intel co-founder, Gordon Moore, famously remarked that the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years. Moore's Law, as it has been called, has proven close to the mark for more than 40 years. The result is that the computers of today are faster, easier to use, more affordable, and more powerful than those of only ten years ago. For instance, a smart phone you might buy today has more memory, storage space, and computing ability than a Cray-1 supercomputer of the 1970s, which sold for over 8 million dollars!

## 1.2 Describing Computation with Language

With this kind of computational power, it might seem anything is possible. But mathematicians in the 1920s and 30s wondered if perhaps some computers could solve problems that other computers couldn't. A young Alan Turing<sup>1</sup> (featured in Figure 14) answered this question in the negative by proposing an ideal computer that we now call a Turing Machine and then proving that this machine could perform any mathematical procedure that could be represented by an algorithm [15]. An important consequence of this work is the linking of computational power or expression with language. If a computation is possible, it can be expressed in language.

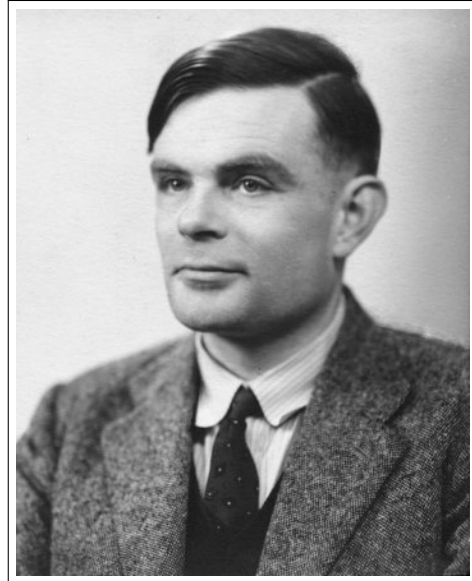


Figure 14: Alan Turing is regarded by many to be the father of computer science.

And furthermore, the language used to express a computational process can be independent of the machine which executes the process.

This idea of machine independent programming was popularized, in part, by Grace Hopper<sup>2</sup> (featured in Figure 15) who, in 1952, coined the word compiler to describe a piece of software that transformed human readable program instructions into machine code [4].

In 1953, John Backus wrote a proposal at IBM to develop **FORTRAN**, which would become the first widely used high-level programming language. Other

---

1. Alan Turing was a World War II code breaker, mathematician, philosopher and computer scientist, but he was also persecuted by the British government for homosexuality. In 2009, Prime Minister Gordon Brown issued a posthumous apology on behalf of the British government for “the appalling way he was treated.” Turing’s influence on the field of computer science was profound and included contributions to computability, computer languages, machine architectures, and artificial intelligence. The Association for Computing Machinery’s (ACM) highest honor is the prestigious **Turing Award**.

2. Grace Hopper’s life and achievements inspired the creation of the **Grace Hopper Celebration of Women in Computing**. This annual conference highlights the research and career interests of women in computing.



Figure 15: *Grace Hopper was a Navy rear admiral who developed one of the first compilers for a computer programming language and helped popularize the idea of machine-independent programming languages.*

influential languages from the 1950s include John McCarthy's **LISP**, Grace Hopper's **COBOL**, and the collaboratively designed **ALGOL**.

Since the 1950s we have seen an explosion of computer languages which explore different paradigms and domains. The language we will use in this text, **Python**, was first released in 1991 by Guido van Rossum.

In some respects the language that a computer uses isn't so different than human language. If you've ever used a cookbook you are probably familiar with procedural languages. A cookbook will, step-by-step, tell you what to do with some ingredients (the input) in order to produce a tasty dish (the output). You can think of a computer program as a list of instructions much like a recipe. But where humans are good at working with ambiguities like "a pinch of salt" or "season to taste", computers are not. Much like legal documents that require carefully crafted language with unambiguous meaning, programming languages require unambiguous instructions that follow strict rules of syntax (legal words and symbols) and semantics (the meaning of those words and symbols). Consider, for instance, this example program which prints a greeting:

```
1 print("Hello_world!")
```

Output:  
Hello\_world!

If we were to leave out even a single quote or parenthesis the structure and meaning of the program changes and it might produce an error instead of the result that we intend. Writing software will require us to be precise and unambiguous. The reason for this linguistic precision is that when a computer evaluates a program it is mapping, or transforming, high level commands to more complicated low level commands that the machine understands.

We often think of programming as a means of communicating instructions to a computer, but if that were our only concern we would simply write programs in machine code made up numerical operator codes and numerical references to registers and memory locations. Consider the simple instruction,

```
1 area = width * height
```

At a low-level a number of things must happen for a computer to do this calculation:

1. the computer must load the value stored at the address associated with width into register A,
2. the computer must load the value stored at the address associated with height into register B,
3. the computer invokes an add operation with A and B,
4. the computer stores the result from A into area.

On an ARM processor each one of these instructions (e.g., load, add, store) has a fixed size numeric code that forms the first part of an instruction and the second part of an instruction identifies a register or memory address that the operand works with. After looking up these instruction codes and memory values we can take these four instructions and write code designed for the computer to execute:

```
1 11100001010100010000001000001010
2 11100001010100010000000000000101
3 11100001100100010000000000000000
4 11100001110100010000000000010101
```

Machine language varies from machine to machine and specifics also vary between operating systems. If we wanted to write this same program in machine

code for a different computer we would essentially have to start over. Working backward from just the machine code, it's impossible to tell that we were calculating the area of a rectangle!

When we write software with high-level programming languages our intent is to communicate our computational descriptions and ideas to ourselves and the broader programming community. Our understanding of a computational process is what gives it meaning. Or as computer scientist Hal Abelson once wrote [1], “programs must be written for people to read, and only incidentally for machines to execute.”

### 1.3 A Problem Solving Strategy

One of the biggest mistakes that a new programmer makes is writing code before he or she fully understands the problem. Computer scientists generally follow a series of software development steps, called a **software development process**, in order to solve problems. In this book we will use a simple four-step approach that works well for small problems and embodies the core principles of almost any software development process:

1. understand the problem,
2. plan your solution,
3. implement and test your solution, and
4. reflect on your solution<sup>3</sup>.

Note that actually writing computer code doesn't take place until you get to step **three**! In fact most of the intellectual heavy lifting happens in steps one and two as we analyze the problem and devise a solution. You might also observe that this is really a general purpose problem solving framework and echoes George Pólya's four step approach to mathematical problem solving [19].

We will break down this process step-by-step so we can start using it. To make our discussion more concrete we will consider the problem of navigating a maze from its start point to its end point. While you might be able to come up with specific instructions to navigate the specific maze drawn in Figure 16, can you devise generalized instructions that someone could use to solve similar mazes?

---

3. Although we have presented this as a linear sequence software development often jumps back and forth between these steps. New problems and requirements often emerge with time creating a software life cycle where software is continuously improved and maintained.

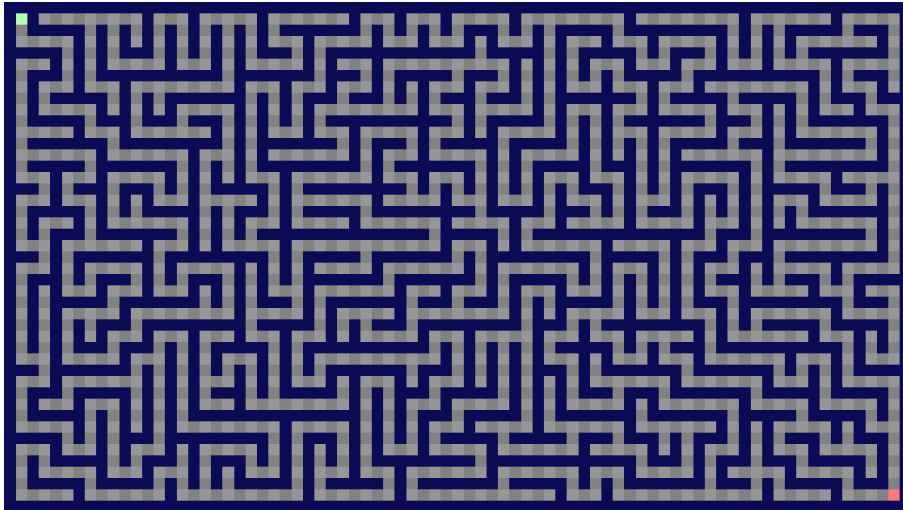


Figure 16: Mazes and labyrinths date back to antiquity and have been important artifacts of art, history, entertainment, and religion. What properties of this maze can help us develop a general maze solving algorithm?

**Step 1** (Understand the Problem). *The first step in the problem solving process is understanding the problem by identifying constraints, enumerating requirements, and asking analytical questions about the problem.*

Many people begin the analysis of a problem with what's called "back of the envelope thinking". This might involve sketching what a system looks like, enumerating relevant equations or knowledge, and outlining relationships. If the problem involves a process we might attempt to work through a few steps of the process in order to understand it. Some of the analytical questions we should be asking ourselves at this stage include:

- Do I understand the problem?
- Can I restate the problem in my own words?
- Do I have the information needed to solve the problem?
- Does the problem have a solution?
- What does an answer to the problem look like?
- Can the solution be verified?



Often times the problems that we work on are complex, under-defined, or involves many people with different goals and expectations. In these cases, understanding the problem may involve collecting user stories or use cases from project stakeholders. We might need to produce formal documents that analyze, specify and validate project requirements. We also might need to use a software development model that is sensitive to projects where the requirements are constantly changing.

Lets consider an example where we really need to understand the problem in order to come up with a solution.

By analyzing the maze in Figure 16, we can discover inherit structures and properties that we can then exploit when developing algorithms to solve the maze. If we don't understand the maze under consideration (e.g., the rules governing how walls and passages are connected together), we could easily come up with a solution that might only work part of the time or even not at all! This particular maze is what is called a simply-connected maze, meaning that all the walls are connected together.

**Step 2 (Plan a Solution).** *The second step in the problem solving process is devising a solution to the problem. This process begins by choosing general problem solving heuristics and then using a variety of design and modeling tools to describe potential solutions to the problem.*

We will usually consider many different potential solutions to a problem so it's critical that our tools for sketching and evaluating potential solutions are quick and light weight. A few common design and modeling tools include:

- writing pseudocode,
- developing mathematical formulas,
- drawing diagrams or flow charts, and
- creating worked solutions for specific instances of the problem.

In this book we strongly advocate that most design and modeling activities should happen on paper or a whiteboard. These mediums let us sketch quickly and flexibly and it's easy to discard ideas. One danger of planning a solution on a computer is that it takes so much time to represent designs formally that it's sometimes hard to discard a bad solution simply because you have so much time invested in developing it!

Executing your solution on paper is important because if you can't see how it works on paper, it's unlikely you will get it working when you set the solution in code.

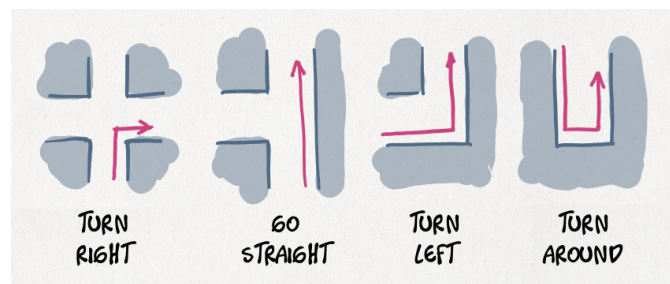
In the coming chapters we will also discuss techniques and strategies for computational problem solving. Choosing appropriate techniques to solve a problem is one of the real arts of computer science and is primarily learned by attempting to solve lots of problems!

One trivial solution to the maze solving problem would be to randomly select a corridor at each branching point in the maze. While this might find a solution, it's also possible that it never finds a solution by randomly walking in circles.

A better solution might exploit the simply-connected nature of the maze in Figure 16. Since all the walls are connected, we should be able to follow the walls themselves to the exit by consistently choosing the left or right hand side. This is known as a wall following algorithm. Figure 17 illustrates the right hand wall following rule, but a left hand rule would also find the exit.

Planning a solution also involves selecting appropriate data structures and computational representations. For example, how is the maze stored in memory by the computer? And how do we interact with it? The solution we have developed depends on following walls, but what if we don't have a list of walls but instead a list of corridors available at each cell in the grid? In planning the solution we often develop pseudocode that sketches the solution strategy. Pseudocode is a free form human readable computation description that doesn't correspond to a formal programming language.

One solution for this problem is to develop a set of instructions that uses the right hand wall following rule based on available corridors. We might start by sketching each kind of corridor decision we might encounter:



At this point in the problem solving process we want to keep potential solutions light-weight so they can be changed or discarded. By sketching the

possible corridor decisions on paper we can identify decisions that don't make sense and test our ideas. In this example, we have graphically identified four unique cases.

Once we are happy with the general strategy, we can write simple rules in pseudocode that describe the strategy:

```
1 until we find the exit:
2   if a right entry exists:
3     turn right and then go forward
4   if a forward entry exists:
5     go forward
6   if a left entry exists:
7     turn left and then go forward
8   otherwise:
9     turn around and then go forward
```

Using these instructions can you make it to the end of the maze pictured in Figure 16? Before we implement our solution in code we need to convince ourselves that the solution will work. This often involves mentally tracing our solution through test cases but could also involve a more formal proof of correctness.

**Step 3 (Implement and Test).** *The third step in the problem solving process is transforming an informal plan into formal testable code.*

While this is a daunting step for the beginning software developer, if you've put in the appropriate work and thinking in steps one and two, this step is just a matter of care and patience. It's a transformation from your informal representation of a problem solution to a formal representation. As we develop our implementation, we also devise tests for our implementation. The tests help us check the correctness of our work and as our system becomes more complex, these tests help us enumerate and enforce problem invariants. We will defer our discussion on testing until Chapter 10.

The next code listing is a formal representation of the informal procedure we developed in the last section. Don't worry if you don't understand everything about the syntax in this example - we'll get there in the next few chapters. The important thing to get out of this example is how writing code is an extension of the computational thinking and planning we did in step two.

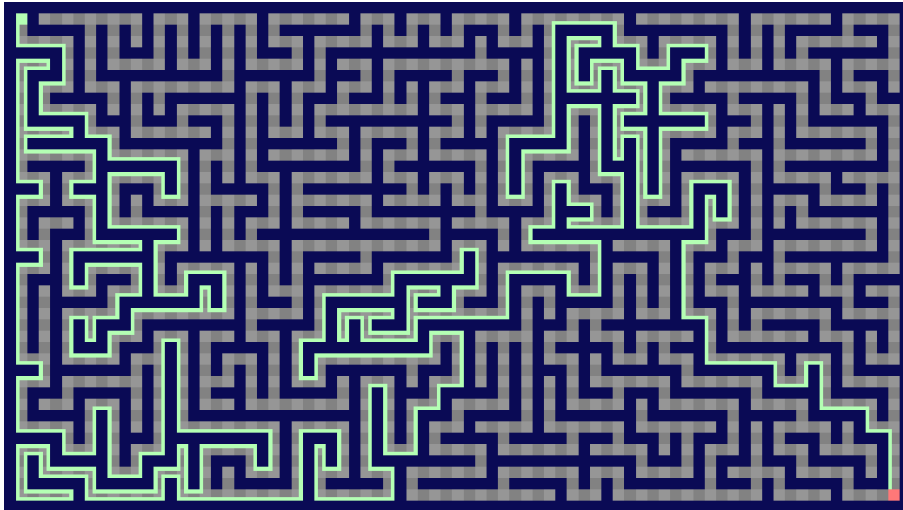


Figure 17: A traversal of the maze using the right hand wall following rule.

[Video](#)

Even without understanding all of the syntax, one thing you might observe is that not all of the text is computer code. Lines that begin with a `#` are comments in the Python language. Comments are human readable code annotations intended to make programs easier to read and understand.

```
1 from pspace.maze import *
2
3 create_maze()
4
5 while not is_exit():
6
7     # The right handed wall following rule has
8     # four cases, which depend on the corridors
9     # available:
10
11     if has_right_entry():
12         turn_right()
13         forward()
14
15     elif has_forward_entry():
16         forward()
```

```
17
18     elif has_left_entry():
19         turn_left()
20         forward()
21
22     else:
23         # turn around!
24         turn_right()
25         turn_right()
26         forward()
```

**Step 4 (Reflection).** *The fourth step in the problem solving process is reflecting on our solution in such a way that we can potentially improve our approach to solving this or other problems.*

In a nutshell, reflection poses the question, “was there a better way to solve the problem?” Thinking deeply about how we have solved a problem helps us design better solutions and identify other similar problems in the future. The kinds of questions you should ask when reflecting on a solution include:

- Is there code that isn’t needed?
- Is there code that is duplicated?
- Can the solution be simplified?
- Is there a different organization that could be used? Why were alternate organizations discarded?
- Is there a different problem solving heuristic that could have been used? What features of the problem or the strategy made one heuristic more attractive than another?
- Could you generalize the solution such that it worked for other problems? What other problems are similar?
- What patterns were important in solving the problem?
- What were the most important design decisions in solving the problem? What alternatives could you have considered?

For the maze solving algorithm we might ask questions such as: Is this the best way to organize the code? Is this the best algorithm for solving a simply-connected maze? What happens if the maze isn’t simply connected?

Another approach due to Charles Trémaux can handle mazes that aren't simply connect by drawing lines on the floor to mark each time you visit a passage [16]. His algorithm is simple: on arriving at a space that has not been visited before (that is, no previous marks), move in a random direction. If you arrive at a marked space and your current path is marked only once then turn around and walk back. If this is not the case, pick the direction with the fewest marks. Trémaux's algorithm dates to the mid 1800s and is an example of a computational process or algorithm that predates digital computers.

There are plenty of other maze solving algorithms that make different assumptions about the structure of the maze, where you start in the maze, or qualities of the generated path. Maze solving can be thought of as a search problem and many of the algorithms we might consider for maze solving will work to solve other search problems and vice versa. Even Trémaux's algorithm is really just a variant of a more general search technique known as depth-first search.

Reflection may also be more narrow in scope. When we reflect on the code itself (and not the entire problem solving process) we are doing something called **refactoring**. Refactoring concerns changing the code of a program without changing its behavior to improve the readability, extensibility, and maintainability of software.

#### 1.4 How to think like a Computer Scientist

Hopefully you are getting the impression that problem solving is a pretty big deal in computer science.

The kinds of analytical reasoning and problem solving that we do in computer science is a mental skill that you will have to nurture if you want to be a computer scientist. And your ability to solve harder or less familiar problems is closely related to your attitudes about problem solving. Positive attitudes position you for success and negative attitudes position you for failure. Consider each pair of positive and negative statements below and ask yourself which statement corresponds most closely with your own attitudes and beliefs:

negative    You either know how to solve problems or you don't.

**positive**    I can solve problems through careful persistent analysis.

negative    Successful problem solvers are lucky.

- positive** Successful problem solvers have strategies for success.
- negative If I don't understand how to solve a problem it's probably someone else's fault for making the problem too hard.
- positive** I'm excited to take on new problems that make me think.
- negative When faced with a new problem I don't know where to start.
- positive** When faced with a new problem I ask questions, I sketch drawings, and engage in a problem solving process.
- negative If something jumps out as the solution that's probably it.
- positive** I take great care to understand the facts and relationships carefully and check my own understanding of a problem.
- negative If I have a project to work on, I'll push them off so I know as much as possible before I start.
- positive** I start the projects as soon as possible and work steadily and incrementally to complete them.
- negative I doubt the instructor can help me.
- positive** Instructors can help you academically and professionally (someday I'll need a recommendation letter).
- negative Due dates always sneak up on me and sometimes I don't even know what is due.
- positive** I regularly check for updates and use a planner, todo list or other tools to stay on track.
- negative I like to do my work while watching television, talking on the phone, or playing games.
- positive** I set aside time to focus on my work in a reduced distraction environment.
- negative If the problem looks too tough I'll search the internet for the answer.

- positive** I try to solve problems myself and only after earnestly attempting the problem will I ask for suggestions or help.
- negative If I can't solve the problem quickly it's probably not going to happen.
- positive** I give myself time to think about the problem. Thinking about the problem in a new place or "sleeping on it" can be helpful.
- negative I dislike Python, I would do better if we were using a different language.
- positive** The specific language we learn is largely irrelevant - this book is about problem solving. The skills I am learning apply to the usage of any programming language.

Computer scientists necessarily have positive attitudes about problem solving. But perhaps when you read the negative attitudes maybe some of them sounded familiar. All of these ethics and attitudes are malleable and you absolutely can re-program yourself. In some cases it's a matter of consciously changing the way you think and in other cases it's a matter of consciously changing your behaviors. Problem solving starts with the problem solver.

### Glossary

- computational thinking** A problem solving method that uses computer science techniques.
- computer** A machine that performs calculations.
- computer science** The systematic study and application of computational processes and systems.
- problem solving** The process of working through the details of a problem to reach a solution.
- Python** A programming language created by Guido van Rossum that emphasizes simplicity and consistency.
- refactoring** The process of changing a program's code without changing its behavior in an effort to reduce complexity or improve readability, reusability, or maintainability.
- reflection** (as part of the problem solving process) The process of thinking about how a problem was solved in order to improve how we solve problems in the future.



**software development process** A series of steps or activities that software engineers engage in to develop software.

**software engineering** A branch of computer science that deals with the professional application of computer science and engineering practice to software development.

**source code** Human readable instructions written in a programming language designed to be compiled or interpreted for execution on a computer.

**semantics** The meaning of words and language.

**syntax** Rules that describe legal words, symbols and grammar for a language.

### **Reading Questions**

- 1.1. Who is regarded as the father of computer science?
- 1.2. When was the word "computer" first used?
- 1.3. Why is studying language an important part of computer science?
- 1.4. Who devised the first programmable mechanical computer?
- 1.5. How should the problem solving process begin?
- 1.6. What is ENIAC?
- 1.7. How do you know if you understand the problem?
- 1.8. In the development process when should we write computer code?
- 1.9. What is syntax?
- 1.10. What is semantics?
- 1.11. Why don't we start the software development process by writing code?
- 1.12. When facing an unfamiliar problem what should you do?
- 1.13. The fact that the number of transistors that can be affordably placed on an integrated circuit doubles roughly every two years is called what?
- 1.14. The 1880 census took how long to calculate by hand?
- 1.15. The 1890 census took how long to calculate using Hollerith's computing machines?
- 1.16. Who is considered the world's first programmer?
- 1.17. What are the four steps of the software development process outlined in this chapter?

- 1.18. List four or five possible careers a computer scientist could pursue.
- 1.19. What are some positive attitudes you have that will make you a successful problem solver?
- 1.20. What are some negative attitudes you may need to overcome to be successful at computer science?

### Exercises

For each of the exercises in this section, you should use the four step problem solving process. For step one, list what you know about the problem - what is stated, what is implied, and what you have derived or assumed. For step two, you should describe a plan and show your work toward a solution. For step three, you should develop a clear statement of the solution. For step four, you should ask yourself reflection questions similar to those discussed earlier in the chapter and, in a few sentences, explore the answers.

#### 1-1. Multiples of five and seven

If we list all of the positive integers below 20 that are multiples of 5 and 7 we get 5, 7, 10, 14, and 15. The sum of these numbers is 51. Find the sum of the multiples of 5 and 7 below 100.

#### 1-2. More multiples

The smallest positive integer with multiples 1, 2, 3, and 4 is 12. What is the smallest positive integer with multiples 1 through 10?

#### 1-3. The monkey and the tree

A monkey attempts to climb a 100 foot high tree. Every minute, the monkey climbs upward three feet but slips back two. How long does it take for the monkey to reach the top?

#### 1-4. Two = one?

The following mathematical proof seems to show that two equals one. What's wrong with it?

$$\begin{aligned}
 a &= b \\
 aa &= bb \\
 aa - bb &= ab - bb \\
 (a + b)(a - b) &= b(a - b) \\
 a + b &= b \\
 a + a &= a \\
 2a &= a \\
 2 &= 1
 \end{aligned}$$

**1-5. The missing dollar**

Three men stay at an inn for the night. The innkeeper charges thirty dollars for a room. The men rent one room; each pays ten dollars. Later, the innkeeper discovers he has overcharged the men and asks the bellhop to return five dollars to them. On the way upstairs, the bellhop realizes that five dollars can't be evenly split among three men, so he decides to keep two dollars for himself and return one dollar to each man.

At this point, the men have paid nine dollars each, totaling 27 dollars. The bellhop has two dollars, which adds up to 29 dollars. Where did the thirtieth dollar go?

**1-6. Who won?**

Charles and Ada were excitedly describing the results of the ACM programming competition. There were three contestants, Adam, Joan, and Linus. Charles reported that Adam won the competition, while Joan came in second. Ada, on the other hand, reported that Joan won the competition, while Linus came in second.

Neither Charles nor Ada has given a correct report. Each has related one true statement and one false statement. What was the actual ranking of the three contestants?

**1-7. The wrong sock**

Charles' sock drawer contains five pairs of white socks and seven pairs of black socks. If, without looking in the drawer, Charles takes one sock from the drawer at a time, how many socks must he take before he's guaranteed to have at least one matching pair?

**1-8. Twelve gold coins**

Charles receives twelve gold coins but one of them is fake. The real coins have the same weight but the fake coin's weight is different. Fortunately Charles has a balancing scale with which to compare the weights of the coins. What is the minimum number of weighings required to discern which coin is fake?

**1-9. Single elimination<sup>4</sup>**

121 tennis players participate in an annual tournament. The tournament champion is chosen using single elimination brackets. That is, the 121 players are divided into pairs, which form a match. The loser of each match is eliminated, and the remaining players are paired up again, etc. How many matches must be played to determine a tournament champion? Can you devise a general formula for calculating the number of matches required based on the number of players?

---

4. The structure formed by connected brackets in an elimination tournament is a **binary tree**, an important organizational structure used in numerous algorithms.

**1-10. The hedgehogs<sup>5</sup>**

Two hedgehogs and their two children come to a river [12]. The hedgehogs find a log with which to paddle across the river but it can only hold the weight of one adult hedgehog before sinking. If the young hedgehogs weigh half as much as the adults, how can the hedgehog family cross the river?

**1-11. The fox, goose, and a bag of beans**

Once upon a time a farmer went to the market to purchase a fox, a goose, and a bag of beans[20]. On his way home, he is forced to cross a river by boat. Unfortunately, the boat only has enough room for the farmer and one of his purchases. Complicating the farmer's dilemma is that if the fox is left alone with the goose, the fox will eat the goose, and if the goose is left alone with the beans, the goose will eat the beans. How can the farmer cross the river without losing any of his purchases?

**1-12. Missionaries and cannibals<sup>6</sup>**

Three missionaries and three cannibals must cross a river using a row boat which can carry at most two people [20]. Complicating the trip is that if the cannibals outnumber the missionaries on either bank (including the boat party), the cannibals will eat the missionaries. How can the missionaries and cannibals cross the river without tragedy?

**1-13. Knights and pages**

Three knights and three pages are on a quest, when they come to a river [6]. The knights soon find a small rowboat but it can accommodate at most two people per trip. Complicating the matter is the fact that any page left in the company of another knight, without his own knight to protect him, would die of fright. How can the knights and pages cross the river without incident?

**1-14. A hundred pigs**

A trader buys 100 pigs for 100 pence [12]. For each boar he pays 10 pence; for each sow he pays 5 pence; and for each couple of piglets he pays one pence. How many boars, sows, and piglets did he buy?

**1-15. A fork in the road**

Charles is traveling to a city when he comes to a fork in the road [22, 10]. Because he is unfamiliar with this country he doesn't know which road to

---

5. **River crossing puzzles** tend to be of particular interest to computer scientists because they are a combination of constraints and procedures. The hedgehog problem dates back to at least the 8th century and puzzles like it may be far older.

6. The missionaries and cannibals problem is famous within the artificial intelligence community because it was the subject of the first paper that proposed an analytical approach to problem formulation. Formulating the problem precisely and implementing a program to find the solution remains a popular exercise for AI students.

take. Standing at the fork are two men. Next to the men is a sign which states that one of the two men is a knight (who always tells the truth) and one of the men is a knave (who always tells lies). The sign went on to say that if travelers valued their lives they should ask no more than one man one yes or no question.

What question could Charles pose to find his way to the city?

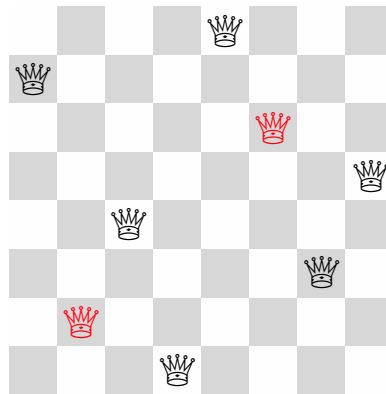
### 1-16. The Tower of Hanoi<sup>7</sup>

The Tower of Hanoi is a mathematical game where three rods are used to hold a number of disks of different sizes, which may slide up and down the rods forming a stack [10]. The puzzle starts with the discs stacked on the first rod in ascending order. The objective is to move the tower of disks to another rod while obeying these rules: 1) only one disk can be moved at a time, 2) the disk must be at the top of the stack to be moved, 3) the disk can only be moved to another stack if it can be placed on a larger disk. Can you solve the problem for three disks? Can you generalize your solution for more than three disks?



### 1-17. Eight queens<sup>8</sup>

Given a standard chess board, can you place eight queens on the board such that no queen threatens another? The solution to this puzzle is not unique; there are 92 possible solutions [3, 8]. In the sample below, shows an incorrect solution where two queens in red threaten each other.



7. The Tower of Hanoi problem appears in almost any introductory text on data structures and variations of this problem continue to intrigue mathematicians and computer scientists.

8. The eight queens problem is famous in computer science for being an early example used to illustrate structured programming [8] and remains a popular recursion and data structures exercise. A common generalization of this problem is the  $n$ -queens problem, which asks for the unique solutions where  $n$  queens do not threaten each other on an  $n \times n$  board.

**1-18. Two generals<sup>9</sup>**

Two armies are preparing to attack a fortified city at the bottom of a valley [2]. The city separates the two armies such that if the generals of each army wish to communicate, they must send a messenger through the valley. Since the valley is occupied by the city's defenders, there is a chance the messenger will be captured.

While the two generals have agreed that they will attack, they have not agreed on a time for attack. In order to succeed, the two armies must attack at precisely the same time. The generals must communicate to each other a time to attack and acknowledge that time. If messengers might be captured both when sending attack times and acknowledgements, how many messages are required to come to consensus?

**1-19. Dining philosophers<sup>10</sup>**

Five silent philosophers sit at a table with bowls of rice [14]. A single chopstick is placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat when he has both left and right chopsticks. A philosopher can take one or more chopsticks at any time but only if the chopstick is not being used by another philosopher. Once a philosopher finishes eating, he needs to put down both chopsticks so they become available to others.

Can you design a discipline of behavior such that each philosopher can continuously alternate between thinking and eating without worry of starvation?

**1-20. The traveling salesman problem (TSP)<sup>11</sup>**

A traveling salesman is given a list of cities that he must visit and a table of distances between each pair of cities. The traveling salesman problem asks what the shortest route the salesman might take that begins and ends at the origin city and visits every other city once. Can you solve this specific instance of the traveling salesman problem with cities A, B, C, and D?

---

9. The two generals' problem is a classic computer science problem for illustrating the challenges related to coordinated communication over an unreliable link. It is closely related to the Byzantine generals' problem where a collection of generals must coordinate an attack with the complication that some of the generals may be traitors [18].

10. The dining philosophers problem is a famous example of the challenges of concurrent algorithm design and synchronization.

11. The traveling salesperson problem dates back to at least the 19th century and since the 20th century it has been an incredibly important problem in combinatorial optimization and theoretical computer science.

	A	B	C	D
A	0	2	10	12
B	2	0	4	8
C	10	4	0	3
D	12	8	3	0

## Chapter Notes

This chapter describes how computational machinery evolved into digital computers and how people like Alan Turing and Grace Hopper made links between language and computation. This chapter also describes a computational solving problem process, which involves:

1. understanding the problem,
2. planning a solution,
3. implementing and testing the solution, and
4. reflecting on the solution.

Problem solving is a skill that can be learned and honed but it takes practice and a positive mental attitude.

We encourage the reader to pay special attention to the problems given at the end of this chapter. Many of these problems are classic computer science problems, which you may encounter again in the context of learning about data structures, algorithms, operating systems, networks, graphics and artificial intelligence. These problems are presented here as a preview of the kinds of exciting computational problems ahead.

- [1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. **Structure and Interpretation of Computer Programs**. The MIT Press, Cambridge, Massachusetts, second edition, September 1996. [isbn: 9780262510875](#).
- [2] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber. Some constraints and tradeoffs in the design of network communications. In **Proceedings of the Fifth ACM Symposium on Operating Systems Principles**, pages 67–74, New York, 1975. ACM. [doi: 10.1145/800213.806523](#).
- [3] W. W. Rouse Ball and H. S. M. Coxeter. **Mathematical Recreations and Essays**. Dover Publications, New York, thirteenth edition, April 2010. [isbn: 9780486253572](#).

- [4] Kurt W. Beyer. **Grace Hopper and the Invention of the Information Age**. The MIT Press, Cambridge, Massachusetts, first edition, July 2009. [isbn: 9780262013109](#).
- [5] Morris Bishop. **Pascal, the Life of Genius**. Reynal & Hitchcock, New York, 1936. [isbn: 0837100216](#).
- [6] Yuri Chernyak and Robert M. Rose. **The Chicken from Minsk: And 99 Other Infuriatingly Challenging Brain Teasers from the Great Russian Tradition of Math and Science**. Basic Books, New York, April 1995. [isbn: 9780465071272](#).
- [7] Douglas Crockford. **JavaScript: The Good Parts**. O'Reilly Media, Inc., Sebastopol, CA, 2008. [isbn: 0596517742](#).
- [8] Edsger Wybe Dijkstra, C. A. R. Hoare, and Ole-Johan Dahl. **Structured Programming**, pages 72–82. Academic Press, New York, first edition, February 1972. [isbn: 9780122005503](#).
- [9] J. Fuegi and J. Francis. Lovelace & Babbage and the creation of the 1843 ‘notes’. **IEEE Annals of the History of Computing**, 25(4):16–26, October 2003. [doi: 10.1109/MAHC.2003.1253887](#).
- [10] Martin Gardner. **Hexaflexagons and Other Mathematical Diversions: The First ‘Scientific American’ Book of Puzzles and Games**. University of Chicago Press, Chicago, second edition, September 2008. [isbn: 9780521756150](#).
- [11] Herman H. Goldstine. **The Computer from Pascal to von Neumann**. Princeton University Press, September 2008. [isbn: 1400820138](#).
- [12] John Hadley and David Singmaster. Problems to sharpen the young. **The Mathematical Gazette**, 76(475):pp. 102–126, 1992. [issn: 00255572](#). [link](#).
- [13] D. S Halacy. **Charles Babbage, Father of the Computer**. Crowell-Collier Press, 1970. [isbn: 0027413705](#).
- [14] C. A. R. Hoare. **Communicating Sequential Processes**. Prentice-Hall, Inc., Upper Saddle River, NJ, 1985. [isbn: 0-13-153271-5](#).



- [15] Andrew Hodges and Douglas Hofstadter. **Alan Turing: The Enigma**. Princeton University Press, centenary edition, May 2012. [isbn: 978-0691155647](#).
- [16] Édouard Lucas. **Récréations mathématiques**. Gauthier-Villars, Paris, 1882. [link](#).
- [17] Scott McCartney. **Eniac: The Triumphs and Tragedies of the World's First Computer**. Walker & Company, New York, June 1999. [isbn: 9780802713483](#).
- [18] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. **Journal of the ACM**, 27(2):228–234, April 1980. [issn: 0004-5411](#). [doi: 10.1145/322186.322188](#).
- [19] George Pólya. **How to Solve It: A New Aspect of Mathematical Method**. Princeton University Press, second edition, October 1973. [isbn: 9780691080970](#).
- [20] Ian Pressman and David Singmaster. “The jealous husbands” and “The missionaries and cannibals”. **The Mathematical Gazette**, 73(464):73–81, 1989. [issn: 00255572](#). [doi: 10.2307/3619658](#).
- [21] T. R. Reid. **The Chip : How Two Americans Invented the Microchip and Launched a Revolution**. Random House Trade Paperbacks, New York, revised edition, October 2001. [isbn: 9780375758287](#).
- [22] Raymond M. Smullyan. **What Is the Name of This Book?: The Riddle of Dracula and Other Logical Puzzles**. Dover Publications, Mineola, NY, July 2011. [isbn: 9780486481982](#).
- [23] Jeannette M. Wing. Computational Thinking. **Communications of the ACM**, 49(3):33–35, March 2006. [doi: 10.1145/1118178.1118215](#).
- [24] Niklaus Wirth. The programming language Pascal. 1973. [isbn: 3-540-43081-4](#). [link](#).



## 2

# Getting Started with Python

*Python is a replacement for the BASIC language in the sense that Optimus Prime is a replacement for a truck.*

*MFen of freenode.net*

```
1 overview = ["Why Python?",
2             "Installing Python",
3             ["Windows",
4             "Windows Annoyances",
5             "OS X",
6             "Linux"],
7             "Running Python",
8             ["IDLE",
9             "The Command Line"],
10            "Programs",
11            "Comments",
12            "Glossary",
13            "Reading Questions",
14            "Exercises"
15            "Chapter Notes"]
```

In this chapter we introduce Python and provide our rationale for why it's a good first language. We'll describe how to install Python, use Python to evaluate Python source files and also describe some of the other Python implementations

that exist.

## 2.1 Why Python?

The official Python documentation describes Python with these words:

**Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.**

There's a lot to like about Python as a first language:

- Python is **simple**. Python's designers have consciously designed the language with a preference for simple English words, special forms that use white space, and a minimalist aesthetic.
- Python is **easy to learn**. Python's syntax and programming conventions are very uniform. By learning only a handful of syntactic constructs and operators you are able to write very sophisticated applications.
- Python is **free**. Python is an example of free software. Not only can you can freely distribute copies of Python, you can also read its source code, make changes to it, and use pieces of it in new free programs.
- Python is a **community**. While Python was originally created by Guido van Rossum, the Python we know today is the effort of millions of developers who participate in improving the language, writing documentation, helping other people learn Python and solving problems with this language in a wide array of domain areas.
- Python is a **high-level language**. When you write programs in Python, you rarely need to bother with low-level processor and architecture specific details. Python abstracts how instructions are executed by the CPU, most memory management chores, and provides syntactic support for common data structures.
- Python is **portable**. Due to its open-source nature, Python has been made to work on many different platforms. You can use Python on GNU/Linux, Windows, FreeBSD, Mac OS, Solaris, OS/2, Amiga OS, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, Vx-Works, PlayStation OS, iOS, Android and many more!

- Python is **object-oriented**. Python is a multi-paradigm language that supports several different programming philosophies. Many older programming languages only support procedural programming, where software is built around reusable programming pieces called procedures or functions. In object-oriented programming, the program is built around objects which combine data and functions in a logical way. Python supports both of these paradigms (and many more!) in a powerful but simple way.
- Python is **extensible**. If you need a critical piece of code to run very fast you can code that part of your program in C/C++ and then call that code from your Python program.
- Python is **embeddable**. You can embed Python within C/C++ programs to allow your program to call Python code. A number of popular applications use Python in this way including **Sublime**, **Blender**, and ArcGIS.
- Python is a **batteries included language**. The Python Standard Library is huge. You can scrape and match text with regular expressions, produce documents in a myriad of different formats, unit test your software, implement threading, access databases, build web servers, send email, traverse XML, develop graphical user interfaces and much more. And all this is always available wherever Python is installed! Besides the standard library, there are even more high-quality libraries which you can install separately.

Bruce Eckel, the author of the well known **Thinking in Java** and **Thinking in C++** books, has this to say about Python:

**I feel Python was designed for the person who is actually doing the programming, to maximize their productivity. And that just makes me feel warm and fuzzy all over.**

Peter Norvig, a well-known artificial intelligence author and Director of Research at Google, has this to say about Python:

**I came to Python not because I thought it was a better/acceptable/pragmatic Lisp, but because it was better pseudocode.**

You might be surprised where you find Python being used. You can find Python powering on-line services by Google and Dropbox, providing infrastructure for commercial games, and providing programmable interfaces for applications like Blender, GIMP, and Inkscape to name just a few. Python really is everywhere.

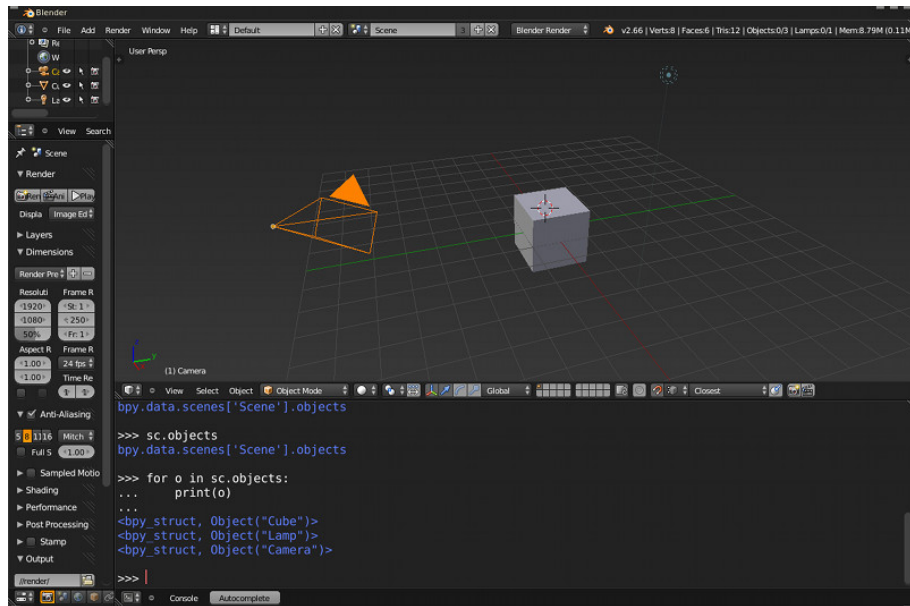


Figure 21: You can easily add new functionality to programs like Blender through Python. Using Python you can modify objects, orchestrate animation, create new 3D operators and even write 3D games!

## 2.2 Installing Python

Python provides installation files for a number of different platforms and it's generally pretty easy to install. There are, however, a few small things that are worth mentioning, so we'll describe how to install Python on Windows, OS X, and Linux. The first thing to note is that this text assume you will be using Python 3.4 or greater. For the best experience we recommend installing the latest stable version.

### Windows

On Windows, your best option is to download the installer directly from the Python website:

<http://www.python.org/download/releases/>

When you begin the installation it will prompt you to enable or disable Python components. Do not uncheck any optional components! And be sure that the

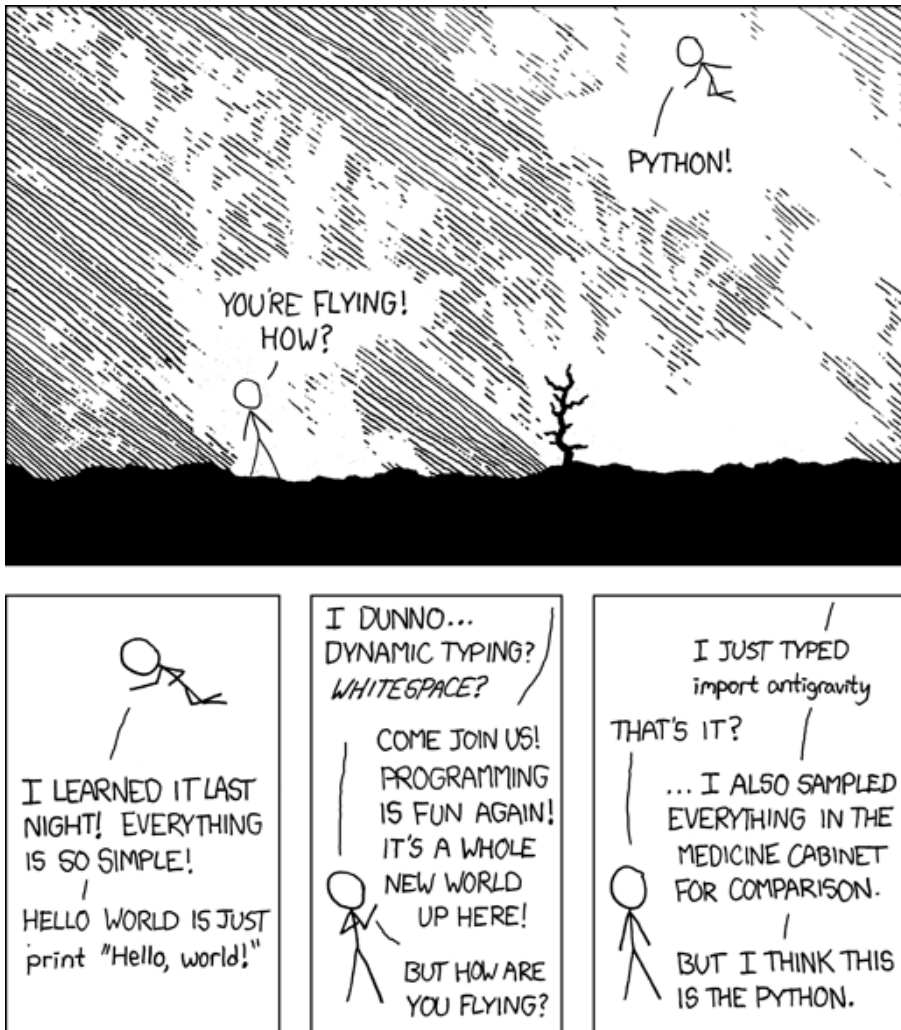
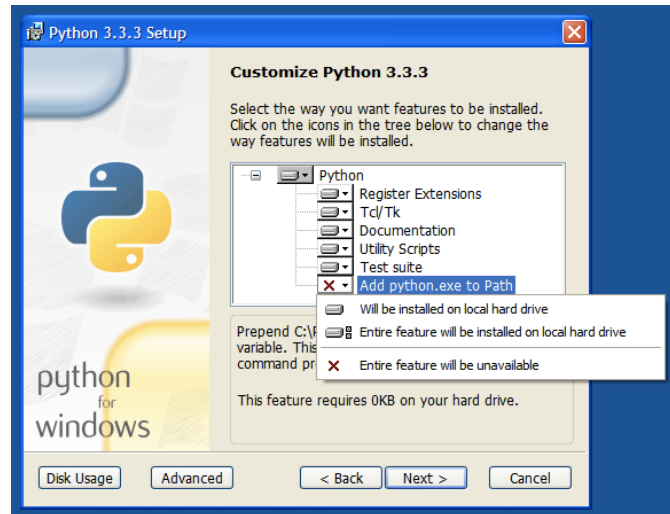


Figure 22: Randall Munroe's *XKCD* featured Python in comic 353. The image rollover reads, "I wrote 20 short programs in Python yesterday. It was wonderful. Perl, I am leaving you." This image is licensed under a Creative Commons Attribution-NonCommercial 2.5 License.

component, “Add python.exe to Path” is enabled by selecting “Will be installed on local hard drive” as show here:



Click “Next” and once the installer finishes, Python should be ready to do some work!

### *Windows Annoyances*

If Python is correctly configured on your system, you can **skip this section**.

But if you get the error, “Command not found” when you run Python from the command prompt then the `PATH` is likely **not** correct and you will need to adjust the path manually. This could happen for several reasons: perhaps you didn’t click “Add python.exe to Path”, or perhaps Python was already installed, or perhaps another program modified the `PATH`. The solution is to add Python’s application directory to the `PATH` environmental variable.

We recommend installing Rapid Environment Editor which provides a much nicer, safer, more consistent environment variable editor than the environment variable editor that comes with Windows. The installer can be downloaded from the Rapid Environment Editor website:

<http://www.rapidee.com/en/download>

Once installed you can update your path by following these steps:

1. Launch Rapid Environment Editor.



2. Click on the System Variable (found in the left pane) called `PATH`.
3. From the menu select Edit → Add Value.
4. Enter `C:\Python34` and then return. **Note: Make sure this is where Python was installed!**
5. From the menu select File → Save.

### OS X

While Mac OS X comes with Python pre-installed it's often quite old. As of this writing OS X is still shipping with Python 2.7. We recommend downloading the Python 3.x installer directly from the Python website:

<http://www.python.org/download/releases/>

The installation is straight-forward but when you run Python from the command line it will default to the the older 2.x Python that comes pre-installed. It's important to invoke Python with the command `python3` instead of `python` to make sure the newest Python is being used.

### Linux

If you are using a recent Debian or Ubuntu release (including Raspbian for the Raspberry Pi) you probably have a fairly recent copy of Python already installed!

If for some reason Python isn't installed, don't worry - it's easy to do. Open the Terminal application and type what appears after the prompt (e.g., the `%`):

```
1 % sudo apt-get install python3
```

Depending on your distribution, IDLE may not be installed. That's easy to remedy by typing this at the terminal:

```
1 % sudo apt-get install idle-python3
```

## 2.3 Running Python

Python programs can be written in any text editor, which saves its text without special markup. And Python itself can be run like any other program. This gives developers a lot of freedom to select the tools they want to use when working with Python.

This text is tool independent and makes no assumptions about what environment you will use to write and evaluate code. We encourage you to try different tools and development environments and find the tools that make you most productive<sup>1</sup>.

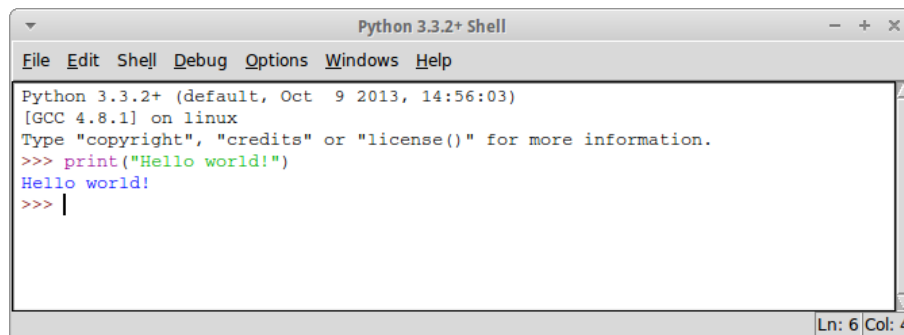
Having said that, Python includes one development environment, which is particularly suitable for new programmers.

### *IDLE*

Python's IDLE (Integrated DeveLopment Environment) is a Python IDE that comes with the Python distribution. IDLE is both an interactive interpreter and a full text editor with syntax highlighting, smart indentation and auto completion.

Launch the IDLE application and you can immediately start working with Python.

Try typing `print("Hello_world!")`. The result should look something like this:

A screenshot of a terminal window titled "Python 3.3.2+ Shell". The window has a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The terminal content shows the Python version and GCC information, followed by the execution of a print statement. The output "Hello world!" is displayed. The cursor is on the next line. The status bar at the bottom right shows "Ln: 6 | Col: 4".

```
Python 3.3.2+ Shell
File Edit Shell Debug Options Windows Help
Python 3.3.2+ (default, Oct 9 2013, 14:56:03)
[GCC 4.8.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello world!")
Hello world!
>>> |
```

You can also use IDLE to work with Python code in files:

1. Select File → New File from the menu.
2. Select the directory where you want to save the file and call it “`hello.py`”.
3. Select Run → Module from the menu.

You should get the same result as the interactive example. If you would like to learn more about IDLE, please refer to the Python Documentation section on [IDLE](#).

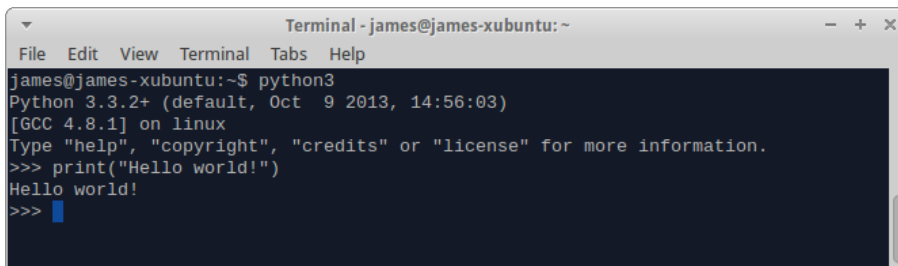
---

1. As a starting point you might look at [PyCharm](#), [Sublime](#), [Komodo](#), [Textmate](#), [Emacs](#) or [Vim](#).

### The Command Line

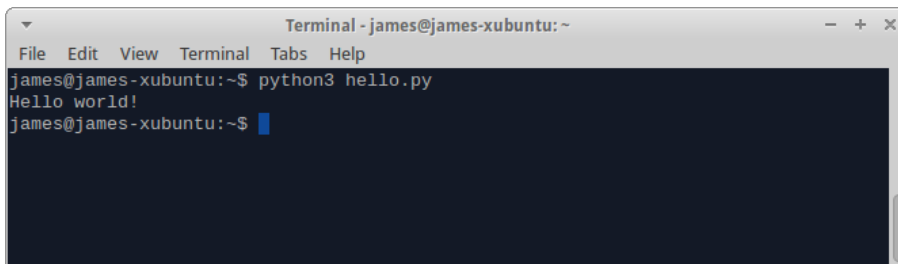
Although you do need a text editor of some sort to write Python programs, you can run Python programs directly from the command line<sup>2</sup> (typically accessed from the Terminal in Linux and OS X or the Command Prompt in Windows). The command line doesn't have a lot of IDLE's bells and whistles but it can be particularly useful if you develop Python programs that are intended to be run as commands. To run python in interactive mode type `python` and then press enter. If you have multiple Python's installed you may need to type `python3`.

Once you see the Python prompt, you can type `print("Hello_world!")`.



```
Terminal - james@james-xubuntu: ~
File Edit View Terminal Tabs Help
james@james-xubuntu:~$ python3
Python 3.3.2+ (default, Oct 9 2013, 14:56:03)
[GCC 4.8.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world!")
Hello world!
>>>
```

If you have save your code in a file you can tell python to execute the code in the file from the command line. In this example I've put the instruction `print("Hello_world!")` in a file called `hello.py`:



```
Terminal - james@james-xubuntu: ~
File Edit View Terminal Tabs Help
james@james-xubuntu:~$ python3 hello.py
Hello world!
james@james-xubuntu:~$
```

We were deliberate when we selected the file name `hello.py` - the dot followed by the two characters “`py`” (called a **file extension**), indicate that this is a Python file, which can be evaluated by a Python interpreter. If you don't include the `py` extension you may run into issues using the file with text editors and integrated development environments.

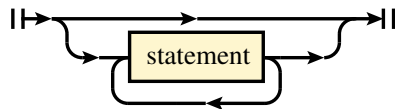
---

2. While it's outside the scope of this text, basic command line literacy is especially useful for software developers. William Shotts' "The Linux Command Line" [29] is an excellent place to start and is released under a Creative Commons license. Most of the material in the book also applies to OS X which, as a Unix derived operating system, has a lot in common with Linux.

## 2.4 Programs

**Programs** are simply a series of statements<sup>3</sup>. Python evaluates these statement one after another. When all the statements have been evaluated the program ends (this is also called exiting). The program may also end if we explicitly tell it to end or the program encounters an error.

**program**



Our “Hello world” program, for example, consists of a single statement - a function call to `print()`, which gives us a mechanism to output or display a value. When each function call is on its own line it becomes a statement.

This program consists of four statements, each of which displays a different value:

```

1 print(1.618)
2 print(7)
3 print(True)
4 print("I am learning computer science!")

```

Output:

```

1.618
7
True
I am learning computer science!

```

The second function we will introduce is the `input()` function. This gives us a way to ask the person using our programs to enter text, which we can capture in a variable.

```

1 value = input("Type your name: ")
2 print("Your name is " + value)

```

---

3. Over the course of this text we will introduce bits of Python grammar using the syntax diagrams described in the Conventions section of the Preface. We also encourage the reader to see where these bits of syntax fit in the larger grammar by periodically skipping ahead to Appendix D.

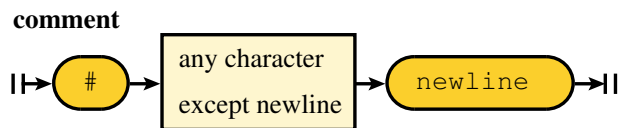
Output :

Your\_name\_is\_James

You might use these two functions quite a bit as you learn to write software but don't become too attached to them. You'll quickly learn other techniques for obtaining data or providing output to the user.

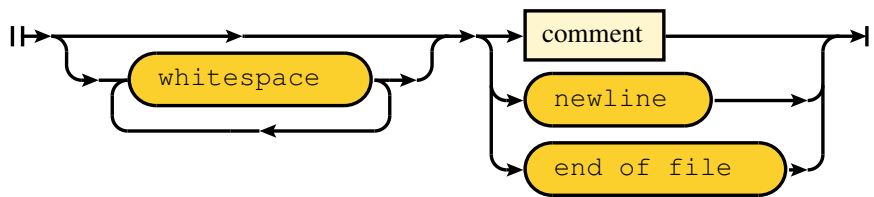
## 2.5 Comments

One other bit of Python syntax we want to introduce early is the comment. A Python comment begins with a hash or pound symbol and continues to the end of the line.



Comments are annotations that are interpreted by Python as a kind of line ending. That is, the actual content of the comment is ignored.

### line end



While comments don't effect the functionality of a program, they do effect our ability to understand software. And it turns out that developers spend an awful lot of time just trying to understand software. In many large software projects more than 90% of a developer's time is spent reading old code.

Sometimes we read old code and have forgotten its purposes. Sometimes we read old code other people have written. And sometimes we work on problems so big we just can't fit the entire solution in our head at one time. Comments provide a way of capturing and transferring the mental models that make solutions possible in the first place.

**Principle 1** (Comments Should Explain Why). *Computer code is by definition a formal description of **how** a process works. It is not, however, an explanation of the motivation, design decisions or context of the software. Use comments to provide this missing context to improve the readability and maintainability of your code.*

Good comments document design decisions:

```
1 # Bubble sort is used to sort the list since at each
2 # iteration only two elements are out of order and
3 # often by a small amount.
```

Good comments describe motivation:

```
1 # As the character walks through the environment we
2 # want the non-player characters to move as if they
3 # each have places to go and things to do.
```

Good comments give us context not obvious from the code:

```
1 # On Windows (but not OS X or Linux) we may lose the
2 # graphics context and it may need to be
3 # reinitialized.
```

Good comments give us context about the problem or our solution:

```
1 # When a user bids on an item and is then outbid,
2 # several things should happen:
3 #
4 # * We should check if their bid should be
5 #   automatically increased.
6 # * We should notify the user via email that they
7 #   have been outbid.
8 # * We should notify the user via a notification if
9 #   they are logged into our website.
```

Bad comments state the obvious or don't advance our understanding of the problem:

```
1 # We use a float to store the Golden Ratio.
2 golden_ratio = 1.618
```

Bad comments are outdated or incorrect:

```
1 # We don't support Facebook
2 facebook_support = True
```

Bad comments are overly terse or incomplete:

```
1 # better than mergesort
```

Consider this comment found in real code:

```
1 # FIX ME
```

Something appears to be broken but there is not enough context to understand what, if anything, is wrong. This could be a legitimate (but incomplete) warning or it could be stale and irrelevant.

Bad comments are overly verbose or don't get to the point:

```
1 # Jane was working on this code in mid December. I
2 # was at the office party when she texted me that the
3 # website had gone down. We didn't know what the
4 # problem was but..
5 # .. 100 lines later ..
6 # and needless to say we were happy with the solution!
```

The software that you write is a story that you are telling other developers. As you begin writing software we encourage you to consciously think about your audience and think about the message you are trying to communicate<sup>4</sup>.

## Glossary

**command line** A text based interface for working with files and commands. On Windows this is accessed through the Command Prompt (derived from the DOS Prompt) and on OS X and Linux the command line is accessed through the Terminal application. This is sometimes called the shell.

**comments** Human readable code annotations intended to make programs easier to read and understand.

**file extension** The ending letters that follow a period in a file name, which suggests which kind of file it is. Audio files in MP3 format, for example, typically end with “.mp3”. Python language files end with “.py”.

**IDLE** Python's default integrated environment.

**input()** A function used to get user input.

**PATH** The name of a special environmental variable that lets the operating system know which directories have executable commands.

---

4. For further reading see Jef Raskin's essay "Comments are more important than code" [27], and Donald Knuth's essay "Literate Programming" [26].

**print()** A function used to output a value or result from computation.

**program** A series of statements which describe instructions intended for evaluation by a computer.

### Reading Questions

- 2.1. What file extension do Python files use?
- 2.2. What is IDLE?
- 2.3. Does Python come pre-installed on Windows?
- 2.4. Does Python come pre-installed on OS X?
- 2.5. Can Python only be run interactively?
- 2.6. What are some philosophical goals of the Python language?
- 2.7. What application gives you access to the command line on Windows?
- 2.8. What application gives you access to the command line on OS X?
- 2.9. Why does Python need to be in the PATH environmental variable?
- 2.10. Name some platforms that Python can be run on.
- 2.11. What does “batteries included” mean with respect to Python?
- 2.12. What benefit does Python being embeddable offer?
- 2.13. Why is the language called Python?
- 2.14. How is Python different from C or C++ with respect to its execution model?
- 2.15. Which version of Python does this text assume you will be using?
- 2.16. Who originally designed the Python language?
- 2.17. What do `print()` and `input()` do?
- 2.18. When does a program end?
- 2.19. What are comments used for?
- 2.20. What are some characteristics of bad comments?



## Exercises

### 2-1. Hello world<sup>5</sup>

A classic beginning programming assignment displays the greeting, “Hello world!” Type this program to try it for yourself:

```
1 print("Hello_world!")
```

### 2-2. Learn from mistakes

What happens if we make a mistake? While we usually want our programs to work correctly, it’s important to know why programs fail. Make a note of what error you get when you modify the program in problem 1 to give an error by

- a) removing one of the quotes,
- b) removing one of the parenthesis, or
- c) capitalizing the word print.

### 2-3. Computer science is fun!

Write a program that displays the following:

```
1 Computer science is fun!
```

### 2-4. Basho

Write a program that prints this haiku by Japanese poet Matsuo Basho,

```
1 a strange flower
2 for birds and butterflies
3 the autumn sky
```

### 2-5. Telescopes

Write a program that prints this famous quote by Edsger Dijkstra,

```
1 Computer Science is no more about computers
2 than astronomy is about telescopes.
```

### 2-6. Comments

Write a program that prints this quote by Norm Schryer,

```
1 If the code and the comments do not match,
2 possibly both are incorrect.
```

---

5. The first hello world program was developed as part of a tutorial by Brian Kernighan and first widely published in the book, *The C Programming Language* [25]. These simple words have been repeated countless times in hundreds of languages.

**2-7. Text banners I**

Write a program that displays the following:

```

1  CCC  SSS   1
2  C    S     11
3  C     SSS   1
4  C      S    1
5  CCC  SSS  11111

```

**2-8. Text banners II**

Write a program that prints your first name as a text banner similar to the one in the last problem.

**2-9. Asterisk pyramids**

Write a program that prints a pyramid using asterisk characters:

```

1  *
2  ***
3  *****

```

and then write a program that prints a pyramid sideways:

```

1  *
2  **
3  ***
4  **
5  *

```

**2-10. Asterisk circles**

Write a program which uses asterisks to display a circle similar to the pyramid renderings in the last problem.

**2-11. Learn computer science!**

Write a program that asks for the user's name. Then display the user's name followed by, " is learning computer science!"

**2-12. Young Dr. Eliza<sup>6</sup>**

This program pretends to be a psychotherapist. Type the program and evaluate it for yourself:

```

1  print("My name is Dr. Eliza. I'm a")
2  print("psychotherapist. In one word,")
3  fear = input("what are you afraid of? ")

```

---

6. Eliza is a computer program originally written by Joseph Weizenbaum in 1966, which uses primitive natural language processing and templated responses to provide human-like interaction [30].

```
4 print(fear.strip().capitalize() +
5       "? That's awful!")
6 print("I'm afraid I can't help with that.")
```

Dr. Eliza gives up pretty easy in this example. Can you extend this program with more questions and **Barnum statements**?

### 2-13. Not much of an adventure

Write a program that uses `print()` to describe a scene like a haunted mansion or a dungeon and then prompts the player for a cardinal direction (i.e., north, south, west, east) with `input()`. Inform the player that they head in their chosen direction but then describe some calamity that impedes their progress (e.g., they get eaten by a goblin!).

### 2-14. I get by with a little help from my friends

Start Python interactively and then type `help()`. Use Python's help function to explore the `print()` and `input()` commands. Answer these questions about the commands:

- a) Does the prompt string include a trailing newline?
- b) Does `input()` return a string with the trailing newline?
- c) Does `print()` take just one argument or more?
- d) By default `print()` always prints a newline after all other content has been printed. Can this be changed? How?

### 2-15. Read the fine manual (RTFM)

Use a web browser to go to <https://docs.python.org/3/>. Using the quick search field can you find documentation for `input()` and `print()`? How does this documentation differ from the information you found in the last exercise? Which did you find easier to use?

### 2-16. The evolution of Python

Changes to the Python language happen largely through a community peer review process. Proposals that describe how the language should be changed or augmented are called PEPs (Python Enhancement Proposal). A list of PEPs is available at <http://python.org/dev/peps/>.

One change made between Python 2 and Python 3 concerns `print()`. In Python 2, `print()` did not require parenthesis around its arguments because it was a statement. In Python 3, `print()` became a function, which must always include parenthesis around its arguments. Which PEP proposed this change? What was the motivation for changing `print()`?

### 2-17. Calculations in Python

In the next chapter we will describe Python's numeric operators in depth but for the most part addition, subtraction, multiplication, and division

work in a way that should be familiar. Launch Python interactively and type:

```
1 20 + 7 * 5 ←
```

Did Python return the result you expected? Now consider this problem: Grace bought 23 notebooks and 17 pens for \$2.30 each. Write an expression in Python that calculates how much Grace paid.

### 2-18. Loose change

Alan has six times as many dimes as quarters. He sums his money and finds that he has \$7.65. How many quarters and dimes does he have? What expression(s) did you evaluate in Python to arrive at an answer?

### 2-19. A question of area

The area of a rectangle is 25 cm<sup>2</sup>. The width is 2 cm less than the length. What is the width and length of the rectangle? What expression(s) did you evaluate in Python to arrive at an answer?

### 2-20. Lightning strike

When lightning strikes we see the flash before the crash of thunder because light travels much faster than sound. If the time between the strike and the thunder is 7 seconds. How far away was the strike in miles? You may assume the speed of sound is roughly 1100 feet per second and one mile is 5280 feet. What expression(s) did you evaluate in Python to arrive at an answer?

## Chapter Notes

Python is an exciting and powerful language. It has the right combination of performance and features that make writing programs in Python both fun and easy. Python is available on a host of different platforms and installing Python is a snap.

In this chapter you learned how to write and evaluate your first program from Python program using IDLE or the command line. You also learned about two basic Python functions (`print()` and `input()`) for interacting with the user and how we can use comments to document our programs.

[25] Brian W. Kernighan and Dennis Ritchie. **The C Programming Language**. Prentice Hall, second edition, 1988. [isbn: 9780131103627](#).

[26] Donald E. Knuth. Literate programming. **The Computer Journal**, 27(2): 97–111, May 1984. [issn: 0010-4620](#). [doi: 10.1093/comjnl/27.2.97](#).

- [27] Jef Raskin. Comments are more important than code. **Queue**, 3(2):64–65, March 2005. [issn: 1542-7730](#). [doi: 10.1145/1053331.1053354](#).
- [28] Scriptol. List of hello world programs in 200 programming languages, April 2013. [link](#).
- [29] William E. Shotts Jr. **The Linux Command Line: A Complete Introduction**. No Starch Press, first edition, 2012. [isbn: 9781593273897](#).
- [30] Joseph Weizenbaum. **Computer Power and Human Reason: From Judgment to Calculation**. W H Freeman & Co, San Francisco, 1976. [isbn: 9780716704638](#).



## 3

# Primitive Expressions

*All of the books in the world contain no more information than is broadcast as video in a single large American city in a single year. Not all bits have equal value.*

*Carl Sagan*

```
1 overview = ["Variables",
2             "Booleans",
3             "Numbers",
4             "Strings",
5             "Lists_and_Dictionaries",
6             "None",
7             "Comparing_Types",
8             "Glossary",
9             "Reading_Questions",
10            "Exercises",
11            "Chapter_Notes"]
```

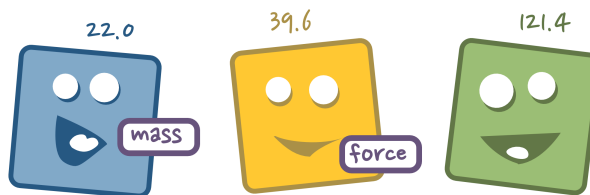
As far as a computer is concerned all information is made up of bits that represent on and off states that we read as ones and zeros. In memory the text that makes up an email message doesn't look much different than the integer pixel values that make up a digital photo. Programming languages distinguish different kinds of information by attaching a type to each value. The type (also called a **data type**) is a formal designation that describes how a value should be stored, what range of values it can take, and what operations

can be performed on it.

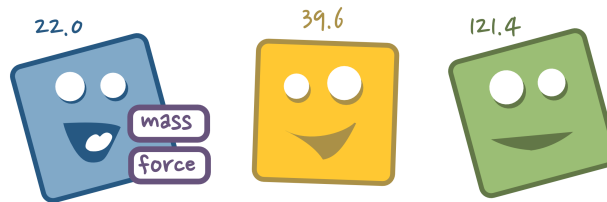
In this chapter, we will discuss variables, values, and core Python types.

### 3.1 Variables

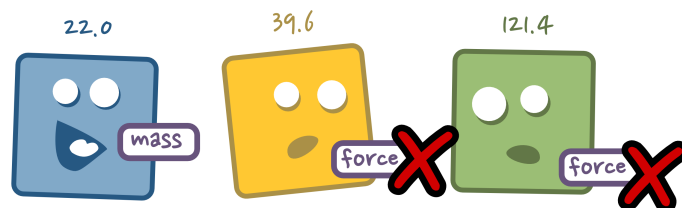
A **variable** connects a name to a value. One analogy that might help you understand how variables names are associated with values is that a variable can be thought of as a sticky label that you might wear to a party. If two values wear their labels we know what to call them but that doesn't mean there aren't other values at the party without a label:



One value could also wear multiple labels - meaning that a single value is connected to multiple variables:



But one important rule at this party is that labels must be **unique** - two values can't have the same label (i.e., variable name) at once:





Variables are connected to values through **assignment** - this is how we make a value wear a label. In Python, assignment is accomplished by placing the equals sign (=) between a target (e.g., a variable name) and an expression that yields a value.

#### assignment statement



We shall demonstrate the assignment syntax by creating a variable called `mass` and assigning it the value `22.0`:

```
1 mass = 22.0
```

In this example we call `22.0` a literal value (or simply a **literal**) because the textual representation of the number directly represents the value. Many of the values that we work with won't be literals and instead will be derived through computation.

Some of the simplest computations we can perform are written as **infix expressions**, where two operands are used by an operator.

You probably learned about infix expressions in the first grade or perhaps even earlier! The expression `1 + 2` is an example where the first operand is `1`, the second operand is `2`, and the operator is a `'+'`. When an operator precedes one or more operands we call this a **prefix expression** (e.g., the negative sign in `-7` is a negation operator that precedes its operand). When an operator succeeds one or more operands, we call this a **postfix expression** (e.g., the exclamation point in `7!` is a factorial operator that succeeds its operand<sup>1</sup>).

We can combine assignment statements and infix expressions to form simple calculations. In this example we calculate a value for `force` based on `mass` and `acceleration` using a standard physics relationship:

```
1 mass = 22.0
2 acceleration = 1.8
3 force = mass * acceleration
4 force → 39.6
```

---

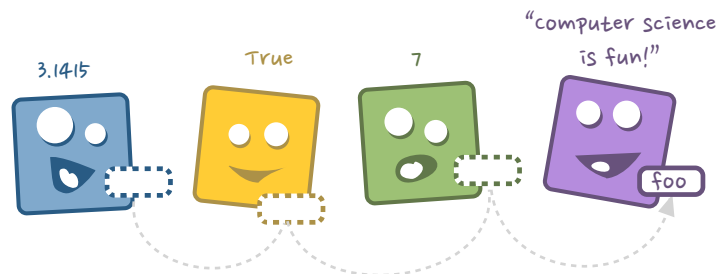
1. As you will discover in the next sections, many Python operators will be familiar from mathematics but not all math expressions are valid Python. `7!`, for example, isn't a valid Python expression.

In this example each variable is of type `float`, a type used for representing real numbers. By assigning `mass` a value of type `float`, Python knows that it can only perform numeric operations on that value and not operations designed for characters, strings, or other kinds of data.

Although Python strongly connects operations to types, types are not strongly connected to variables. A variable's type may change over time in Python. This dynamic relationship between a variable and the type of value it stores is called **dynamic typing**. When the type of a variable is not allowed to change we call this **static typing** (C and Java are examples of statically typed languages). In the next example we illustrate dynamic typing by letting the variable `foo`<sup>2</sup> take on four different values and four different corresponding types:

```
1 foo = 3.1415
2 foo = True
3 foo = 7
4 foo = "Computer science is fun!"
```

With each assignment `foo` is connected to a different value in memory:



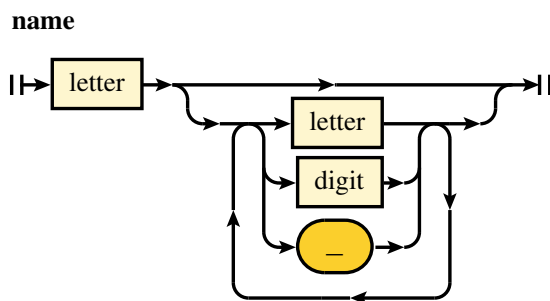
Programming languages have special rules for naming variables such that they won't be confused with literal values or other language constructs. The names that a program uses for variables or other entities are called **identifiers**. In Python, the rules for creating an identifier require that:

- the first character must be a letter of any case or language or an underscore,

---

2. `foo` is a "nonsense" variable name commonly used in Computer Science examples. Its utility is that it strips specific context from the example. According to the New Hacker's Dictionary other "metasyntactic" variable names include `bar`, `baz`, `qux`, `quux`, `corge`, `grault`, `garply`, `waldo`, `fred`, `plugh`, `xyzzy`, and `thud`. It should also be noted that the New Hacker's Dictionary isn't always the most reliable source of information.

- continuation characters may include most characters (e.g., letters, digits, and underscores) but not whitespace or punctuation that has special meaning in Python (e.g., period, comma, parenthesis, brackets),
- identifiers are case-sensitive (e.g., `cat` and `Cat` are recognized as different identifiers), and
- identifiers can't have the same name as words reserved by the Python language as a **keyword** (e.g., `and`, `if`, `def`, and `else`).



There are a little over 30 different keywords to avoid in creating an identifier. The full list can be found in the Python Documentation section on [keywords](#). Breaking any of these rules can result in odd errors or unintended results. Beyond these rules there are also several naming conventions that are not strictly enforced by the Python evaluator but are considered good Python style (see also [PEP8](#)):

- variable names are lower case and words are separated with underscores (e.g., `standard_deviation`),
- class names are capitalized camel case (e.g., `ColorMatrix`),
- identifiers that begin with one or more underscores have special meaning that we will discuss later,
- identifiers shouldn't have the same name as built-in identifiers (e.g., `int`, `float`, `list`, `tuple`, `dir`).

There are also quite a few built-in identifiers to avoid. The full list can be found by running Python interactively and evaluating, `dir(__builtins__)`.

Even with these additional restrictions, developers have a lot of freedom in devising variable names. And unfortunately it's all too easy to choose "bad"

variable names that make code unnecessarily hard to understand. When you are creating a new variable, here's a few questions you might ask yourself:

- Is the name descriptive?
- If you had seen this variable for the first time would the name make sense?
- Is the name too wordy, long, or redundant?
- Is the name too short or does it use uncommon abbreviations?
- Is the name consistent with existing naming conventions?
- Does this value have important units (grams, meters, moles, etc.) that are not obvious from its type or usage?
- Does the name unnecessarily use negative logic or other counterintuitive conventions? You should, for example, consider using `is_enabled` instead of `is_not_enabled`.

**Principle 2** (Code Should Read Like Poetry). *Code written with good variable names allows the code to be read almost like poetry. Use variable names to tell a coherent story about what a computation accomplishes. Poor variable names hide the meaning of the code.*

Consider this perfectly correct piece of code:

```
1 a = (1/2) * b * c
```

What does it do? The variable names do nothing to help us understand the purpose of the code. Without changing the calculation, we can choose more thoughtful names that reveal the purpose of the code:

```
1 triangle_area = (1/2) * base * height
```

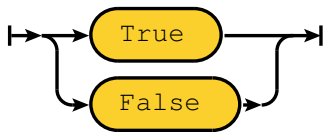
Even without a single comment, it's clear that this small piece of code is calculating the area of a right triangle.

With the last principle in mind, we feel obligated to make the reader aware of some slight hypocrisy on our part; most of the tables in this text that summarize operations or functions will use `x`, `y`, and `z` as variable names as a matter of consistency. But unless you are working with coordinates in space that correspond to `x`, `y`, and `z` you should choose better variable names in your own code.

### 3.2 Booleans

Python uses the `bool` type to supports values that can only be `True` or `False`.

#### bool literal

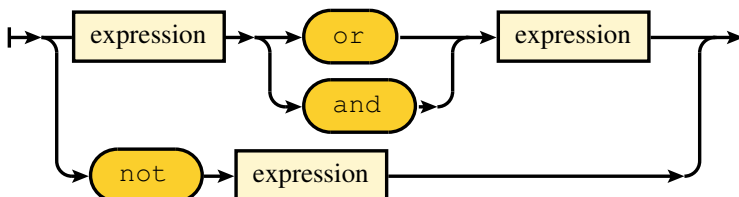


Booleans are used to represent almost any kind of binary state or conceptually opposed values:

- true / false
- on / off
- yes / no
- present / missing
- correct / incorrect
- success / failure

Boolean values can be combined with **logical operators** to yield new Boolean values based on the rules of **Boolean algebra**. Boolean algebra was first described by George Boole in 1847 as a mathematical language for logic [31] and it is fundamental to digital logic and computer programming.

#### logical expression



The core set of Boolean operators Python defines are given in Table 31, which are then both defined and illustrated using the truth table in Table 32.

Table 31: *Logical Operators*

Operation	Description
<code>x and y</code>	The logical conjunction of <code>x</code> with <code>y</code> ( $\wedge$ )
<code>x or y</code>	The logical disjunction of <code>x</code> with <code>y</code> ( $\vee$ )
<code>not x</code>	The negation of <code>x</code> ( $\neg$ )

Table 32: *Logical Operator Truth Table*

<code>x</code>	<code>y</code>	<code>x and y</code>	<code>x or y</code>	<code>not x</code>
False	False	False	False	True
True	False	False	True	False
False	True	False	True	True
True	True	True	True	False

A **truth table** places Boolean parameters and logical expressions in a table such that we can enumerate all the possible combinations of parameters and thus all the possible results of the associated expressions. The truth table can grow in size as we add parameters and expressions. A fully expanded truth table has  $2^n$  rows where  $n$  is the number of parameters. Truth tables provide an excellent tool to practice and visualize Boolean expressions but can also be used to prove logical equivalences and describe logical circuits.

We recommend that Boolean variable names are written as English predicates. When read with logical operators this makes Boolean expressions sound more like English and tends to be less ambiguous. Consider this example that intends to check if a user is both an administrator and online:

```

1 is_administrator = False
2 is_online = True
3 is_administrator and is_online → False

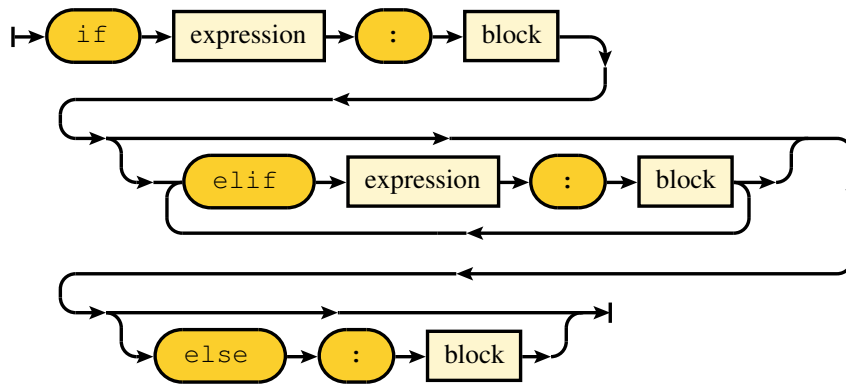
```

Good naming allows the Boolean expression (line 3) to read naturally and without ambiguity.

Boolean expressions can be used to represent conditions like when it is warm and sunny. Python lets us take conditional action using **if statements**. An **if** statement consists of a conditional expression to evaluate, optional **elif** (short

for “else if”) clauses that may be evaluated, and an optional `else` clause that may be evaluated.

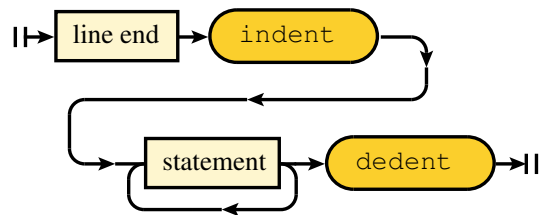
#### if statement



The action to be evaluated when a test evaluates to `True` is described by one or more statements in a **code block** (or **block** for short). Code blocks with multiple statements must be **indented** some amount of space and then **dedented** the same amount of space when the block ends <sup>3</sup>.

As a matter of convention and consistency many Python programmers use **four** spaces to indent code within a block (see also [PEP8](#)).

#### block



`if` statements are the most fundamental tool we have for making decisions in software. Branching happens when we evaluate a piece of code based on a condition that was evaluated. Branching alters the control flow of a program and allows different cases (as determined by the conditional expression) to be associated with different actions.

<sup>3</sup>. Python also allows a single line of code to be placed directly after a colon instead of a block but this syntax is often discouraged.

**Strategy 1** (Break Problems Into Cases). *Many problems can be thought of as having a limited number of unique cases where different actions should be taken. By formally defining the conditions for these cases we can break problems into smaller easier to solve sub-problems.*

Lets demonstrate how an `if` statement can help us decide what to do with our warm and sunny conditions:

```

1 is_sunny = True
2 is_warm = True
3
4 if is_sunny and is_warm:
5     print("Let's go swimming!")
6 elif is_sunny and not is_warm:
7     print("Let's go snowboarding!")
8 else:
9     print("Stay inside!")

```

Once an expression matches one of the `if` or `elif` conditions then the remaining clauses are skipped even if the expression would evaluate to `True`. Consider this example where one of the `elif` clauses will never be displayed:

```

1 is_sunny = True
2 is_warm = True
3
4 if is_sunny:
5     print("I love sunshine!")
6 elif is_sunny and is_warm:
7     print("We should go swimming!")

```

Although it would be nice to go swimming if it is sunny and warm, the last clause in this example will never fire no matter what values are assigned to `is_sunny` and `is_warm`! Alas, we will never go swimming because of our love of sunshine and the use of an `elif`.

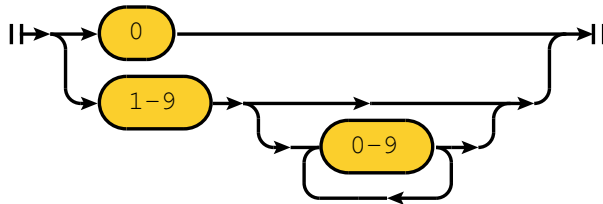
### 3.3 Numbers

Python has two principle numeric types for representing integers and real numbers.



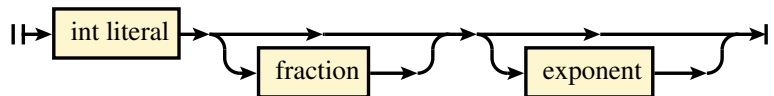
Integers can be represented with the `int` type<sup>4</sup>. The `int` type is used to represent whole or counting numbers and can represent both positive and negative quantities. Python has something called bignum support meaning that the size of an `int` is only limited by available memory [33].

#### int literal

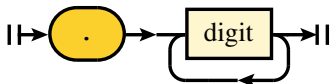


Real numbers can be represented with the `float` type. `float` is short for **floating point** - a method of representing real numbers<sup>5</sup>.

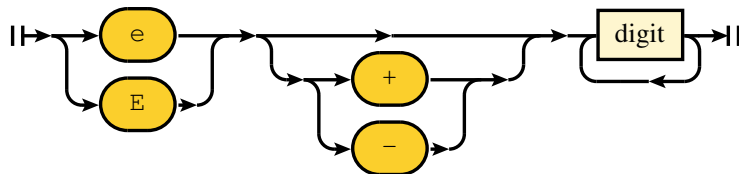
#### float literal



#### fraction



#### exponent



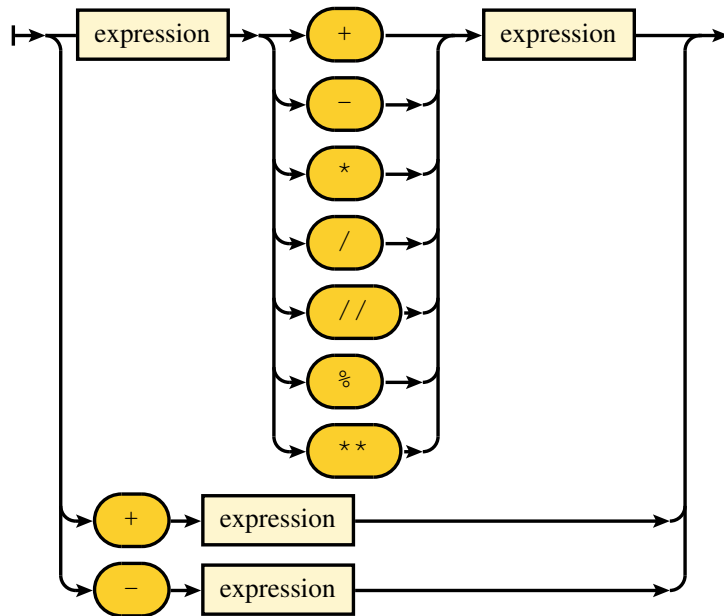
4. The syntax diagram we present here describes only the syntax for base 10 integers. Python can also understand integers written in base 2 (e.g., `0b101010`), base 8 (e.g., `0o52`), and base 16 (e.g., `0x2A`) using base prefixes.

5. Python can also represent imaginary numbers as a composition of two floats separated by a '+' and ending in a 'j' (e.g., `17+3.22j`).

## Arithmetic

Numbers can be combined in arithmetic expressions to produce new numbers.

## arithmetic expression



These operations are described in Table 33.

Table 33: Arithmetic Operators

Operation	Description
$x + y$	The sum of $x$ and $y$
$x - y$	The difference of $x$ and $y$
$x * y$	The product of $x$ and $y$
$x / y$	The quotient of $x$ and $y$
$x // y$	The (floored) quotient of $x$ and $y$
$x \% y$	The remainder of $x / y$
$x ** y$	$x$ to the power $y$
$+x$	$x$ unchanged
$-x$	The negation of $x$

Most operations respect the type such that result will have the same type

as the operands (e.g. adding two `ints` results in an `int`, while adding two `floats` results in a `float`).

Division is an exception. An `int` divided by an `int` yields a `float` as demonstrated in these examples:

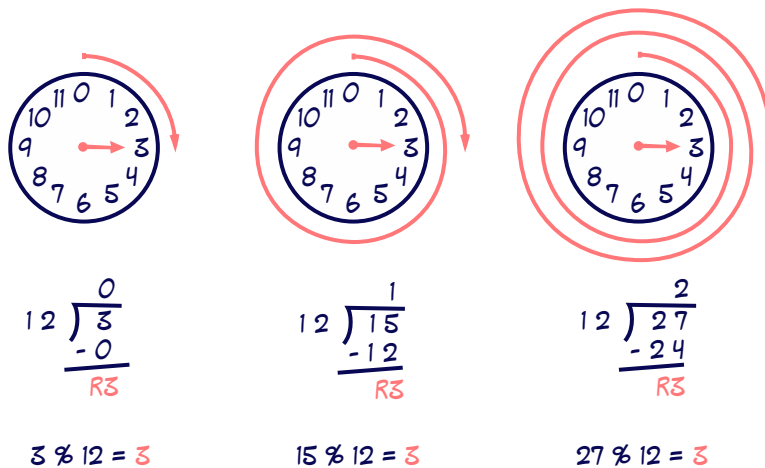
- 1 `8 / 2 → 4.0`
- 2 `7 / 2 → 3.5`

We can also divide numbers under the rules of **integer division** with the `//` operator. Integer division is equivalent to the mathematical floor of normal division. The result will always be an integer in the mathematical sense, but if either operand is a `float` then the return type will also be `float`:

- 1 `8 // 2 → 4`
- 2 `7 // 2 → 3`
- 3 `7.1 // 2 → 3.0`
- 4 `7.1 // 2.2 → 3.0`

Another operator you may be less familiar with is the modulo operator. `x % y` asks the question, “What is the remainder when `y` divides `x`?” Arithmetic that uses the modulo operator is often called **clock arithmetic** because operation results wrap around like a clock after reaching a certain value (the second operand of modulo).

Consider this example where we find `3 % 12`, `15 % 12`, and `27 % 12`:



This turns out to be a pretty useful operation as we will demonstrate in the next example.

Lets say that it is 11 o'clock and we have an appointment in 15 hours. What time is our appointment?

This is easy for us to do on our fingers but how do we express this as a calculation a computer can do? If we add 11 to 15 we get 26, which isn't on an ordinary clock face.

The trick is to somehow let the addition wrap around like a clock. And that's exactly what modulo does:

$$(11 + 15) \% 12 = 2$$

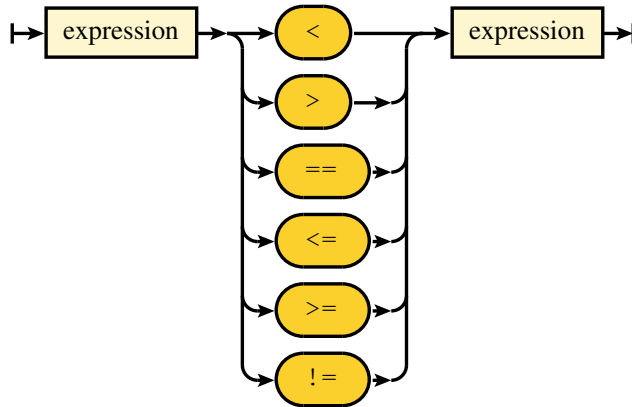
If we divide 26 by 12, we get a remainder of 2 and this gives us our appointment time. Lets generalize this strategy and develop a Python version of this calculation:

```
1 # It's currently 11 O' Clock
2 hour = 11
3
4 # and our appointment is in 15 hours.
5 hour += 15
6
7 # We can use modulo to calculate when our appointment is
8 # as it would appear on a 12 hour clock.
9 hour % 12 → 2
```

Clocks aren't the only thing that wrap around: traffic lights wrap in a sequence green to yellow to red (think modulo 3), in counting money the number of cents goes from 0 to 99 (think modulo 100), and days in a week wrap in a sequence from Monday to Sunday (think modulo 7). Perhaps you can think of some other examples where clock arithmetic can be used?

### Comparison

Numeric values may also be compared with **comparison operations**.

**comparison expression**

Comparison operators differ from the mathematical operators we have discussed so far in that they return Boolean results rather than numerical results. The full complement of comparison operators available in Python are described in Table 34 and correspond to common mathematical inequalities.

Table 34: *Comparison Operators*

Operation	Description
<code>x &lt; y</code>	True if <code>x</code> is less than <code>y</code> ; False otherwise
<code>x &gt; y</code>	True if <code>x</code> is greater than <code>y</code> ; False otherwise
<code>x == y</code>	True if <code>x</code> is equal to <code>y</code> ; False otherwise
<code>x &lt;= y</code>	True if <code>x</code> is less than or equal to <code>y</code> ; False otherwise
<code>x &gt;= y</code>	True if <code>x</code> is greater than or equal to <code>y</code> ; False otherwise
<code>x != y</code>	True if <code>x</code> is not equal to <code>y</code> ; False otherwise

We can combine comparison operators with `if` statements and the logical operators we discussed in Section 3.2 in powerful ways:

```

1 # Our previous example used is_warm but
2 # temperature gives us more precise meaning.
3 if is_sunny and temperature > 70.0:
4     print("Go outside and play!")

```

```

5 else:
6     print("Stay inside!")

```

Accuracy and precision are important concepts when using the `float` type because, unlike Python's `int`, the `float` has a fixed number of bits which are used in its underlying representation. Precision concerns the tightness or tolerance of a specification. Accuracy concerns the correctness of a value. `float` accuracy is a tricky issue because the data type is probably not implemented the way that you might expect.

Many of the values we use in every day calculations are expressed as base 10 decimals, but `floats` are encoded in base 2. Much like the fraction  $1/3$  can't be exactly represented exactly with a limited precision base 10 decimal (it becomes 0.333... repeating), the fraction  $1/10$  can't be exactly represented with a limited precision binary decimal (it becomes 0.0001100110011... repeating). It's a mistake to think floating point numbers are inherently inaccurate, but you should be aware that floating point numbers often correspond to limited precision approximations [32].

What might happen if you wrote a payroll application using `floats`?

As part of our software we might calculate a penny tax on six items using multiplication,

```

1 tax1 = 6 * 0.01

```

And then elsewhere in the program we might calculate the same tax using addition,

```

1 tax2 = 0.01 + 0.01 + 0.01 + 0.01 + 0.01 + 0.01

```

You might be surprised to learn `tax1` and `tax2` are not equal!

```

1 tax1 == tax2 → False

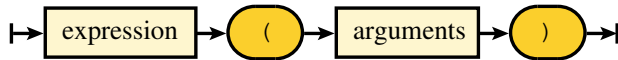
```

Why are `tax1` and `tax2` different? The limited precision of the representation of 0.01 has accumulated enough error over six additions such that 6 times 0.01 does not accurately describe the same value. While Python does have support for arbitrary precision real number types (e.g., `decimal` module, `mpmath` module, `gmpy` module) the `float` is more commonly used (especially when accuracy and precision are limited concerns) because operations on `floats` are relatively fast.

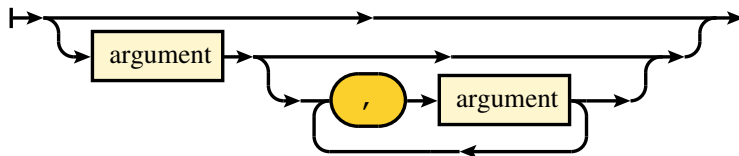
### Function Calls

Some mathematical operations can be invoked by calling a function with a series of arguments.

#### function call



#### arguments



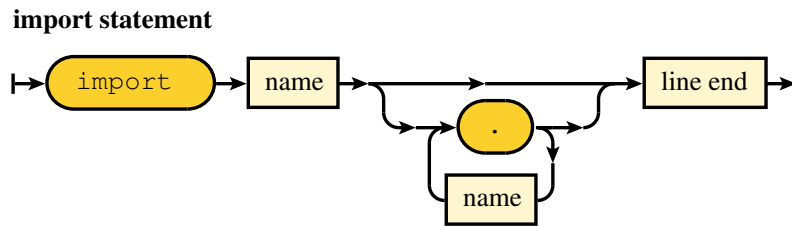
Calling mathematical functions works just like calling the functions `print()` and `input()`, which we introduced in Chapter 2. The built-in numerical functions are summarized in Table 35.

Table 35: *Arithmetic Functions*

Operation	Description
<code>abs(x)</code>	The absolute value or magnitude of <code>x</code>
<code>max(x, y)</code>	The greatest of all arguments; more than two arguments may be provided
<code>min(x, y)</code>	The least of all arguments; more than two arguments may be provided
<code>round(x)</code>	round <code>x</code> to the nearest whole number

### The `math` module

While we have explored some of the most common numerical operations, we are really just scratching the surface of what Python can do numerically. Python has a host of mathematical functions and classes that operate on numbers. Some of the most common operations are located in the `math` module. A **module** is a collection of related functions, classes, and sub-modules that can be included in your code using the `import` statement (typically at the beginning of your code).



Once a module has been imported, we use the dot operator (.) to access variables, functions, classes, or sub-modules contained in the module.

In this example, we import `math` so that we can use the `floor()` and `ceil()` functions.

```

1 import math
2
3 math.floor(12.3) → 12.0
4 math.ceil(12.3) → 13.0

```

For a full description of the functions available in the `math` module, see the Python Documentation on the [math module](#).

#### *The random module*

Not all numeric utilities are found within `math`. The `random` module, for example, provides functions for generating random numbers.

```

1 import random
2
3 random.randint(0, 100) → 37

```

It should be noted that, in general, computers have a great deal of trouble creating truly random numbers. Computers are deterministic and discrete by nature and even events within a computer, which you might expect to be random, are often based on patterns. Thus, when we speak of random numbers we really mean pseudo-random numbers which are generated by an algorithm to have many of the mathematical characteristics that a random series of uniformly distributed numbers should have.

Random numbers are important in probabilistic algorithms and programs that need elements of nondeterminism. Games, for example, often require nondeterministic elements such that the game has different outcomes each time it is played. The game [rock-paper-scissors](#) wouldn't be very fun if your friends always chose rock!



Here's one way you might write a program to play rock-paper-scissors using random numbers:

```

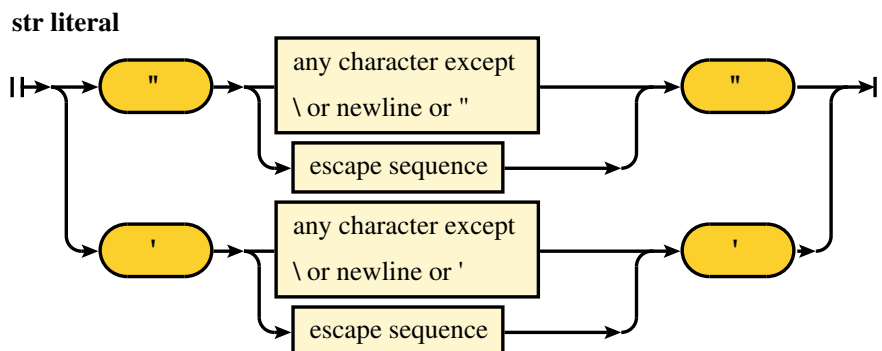
1 import random
2
3 shape = random.randint(0, 2)
4 if shape == 0:
5     print("Rock!")
6 elif shape == 1:
7     print("Paper!")
8 else:
9     print("Scissors!")

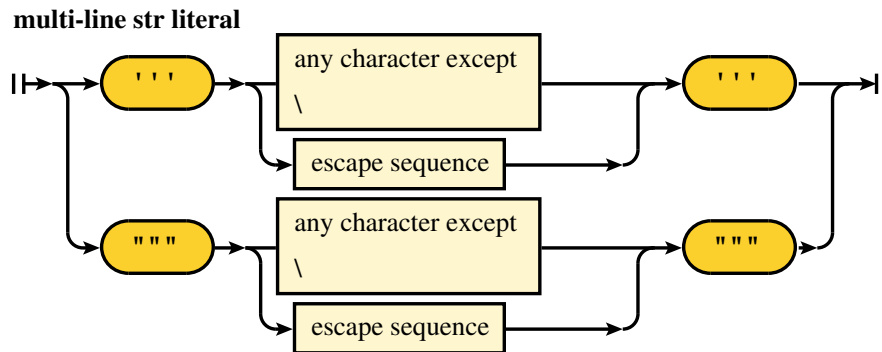
```

For a full description of the functions available in the `random` module, see the Python Documentation on the [random module](#).

### 3.4 Strings

Python can store letters, numbers, and other language glyphs inside a data type called `str` (short for string). Python's string type can not only represent western Latin characters (a-z) and Arabic numbers (0-9) but a host of other foreign language characters and glyphs using a standard called Unicode that maps glyphs to integer representations [34] (see also Appendix C). We combine characters into words and sentences by placing these unicode integers in a sequence we call a string.





We've already encountered several examples of Python strings, most notably the greeting "Hello world!" in Chapter 1. Putting quotes around a sequence of characters forms a string literal but Python has **several** different quoting styles. The different quoting styles **don't** create different kinds of strings, instead they are provided as a linguistic convenience and many Python programmers have adopted informal conventions concerning when one style should be used over another:

Table 36: *Literal String Syntax*

Style	Convention
<code>'hello'</code>	Single quotes are typically used for short "symbol" like strings
<code>"Hello_world!"</code>	Double quotes are typically used for human language messages
<code>"""Hello world!"""</code>	Multi-line quotes (also called triple quotes) can span multiple lines and are used for long strings. They are also used for strings that provide documentation about the program (called doc strings).
<code>'''Hello world!'''</code>	Multi-line quotes can be composed from either thrice repeated single quotes or thrice repeated double quotes

There are a few special character combinations called **escape sequences** that have special meaning within a string literal. Escape sequences begin with a backslash and are followed by one or more characters that make up a valid escape sequence. Some common escape sequences are listed in Table 37 and the full set of escape sequences are listed in the Python Documentation section on [String and Byte literals](#).

Table 37: String Escape Sequences

Sequence	Description
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\n	Linefeed (LF)
\r	Carriage Return (CR)
\t	Horizontal Tab (TAB)
\uxxxx	Character with 16-bit Unicode hex value xxxx
\Uxxxxxxxx	Character with 32-bit Unicode hex value xxxxxxxx

Escape sequences are useful for representing characters that may not be visible or easily represented (like a newline or a tab), embedding quotation, or encoding Unicode characters as demonstrated in these examples:

```

1 # add several newlines at once:
2 print("My First Program\n\nby Alan")
3
4 # separate data items with tabs:
5 print("data1\tdata2\tdata3")
6
7 # embed quotes:
8 print("Press \"enter\" to continue.")
9
10 # write hard to type symbols:
11 print("Smiley face: \u263A")

```

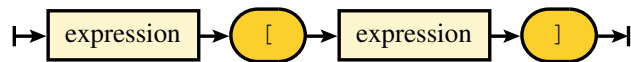
Python strings are implemented as sequences meaning that we can access the individual characters that make up a string. Characters are labeled numerically

in ascending order beginning with zero but we can also access the characters of a string in reverse order in descending order beginning with -1. Thus, while the string “Flagstaff” is nine letters long, its first letter is at index 0 (or -9) and its last letter is at index 8 (or -1):

0	1	2	3	4	5	6	7	8
F	l	a	g	s	t	a	f	f
-9	-8	-7	-6	-5	-4	-3	-2	-1

We use square braces ([ ]) postfix to a sequence expression to access a character by index.

### subscript

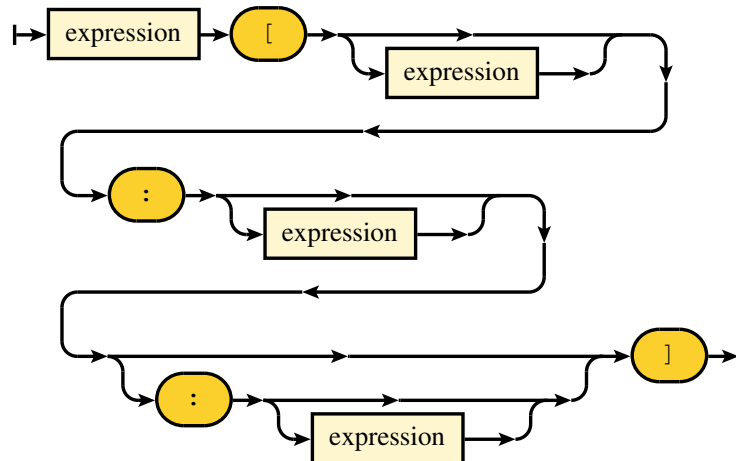


Using the string “Flagstaff” we can pull single characters from the string using the **subscript operator**.

```

1 city = "Flagstaff"
2 city[0] → "F"
3 city[-1] → "f"
4 city[-7] → "a"
  
```

Square braces are also used to create arbitrary substrings from an existing string using the **slice operator**.

**slice**

A slice consists of a start index, an end index, and a step provided in that order. Any expression may also be left empty and a default extent or step is used in place of the empty expression.

```

1 b = "Grace"
2 b[1:3] → "ra"
3 b[2:] → "ace"
4 b[-3:] → "ace"
5 b[:3] → "Gra"
6 b[-1:-5] → ""
7 b[::-1] → "ecarG"

```

Slicing and subscripting also work with other (but not all) Python sequences.

Common operators that work on strings are described in Table 38. While some of the operators have the same representation as the numeric operators described in Table 33, the operation performed is quite different.

Table 38: *String Operators*

Operation	Description
$x + y$	Concatenates $x$ and $y$ and returns the result
$x * i$	Concatenates $x$ with itself $i$ times
$x[i]$	Gives the character at position $i$ in $x$

Operation	Description
<code>x[i:j:k]</code>	Returns a substring of <code>x</code> beginning at index <code>i</code> and ending at index <code>j</code> (exclusive) using step <code>k</code>
<code>x in y</code>	<code>True</code> if the character <code>x</code> is in <code>y</code> ; <code>False</code> otherwise
<code>x not in y</code>	<code>True</code> if the character <code>x</code> is not in <code>y</code> ; <code>False</code> otherwise
<code>len(x)</code>	Returns the number of characters in <code>x</code>

While less familiar than many of the numeric operations, Python's string operators are relatively intuitive and we will use them quite a bit in this text.

```

1 a = "alan"
2 b = "grace"
3 a + b → "alangrace"
4 a * 5 → "alanalanalanalan"
5 "n" in a → True
6 "x" not in a → True
7 len(a) → 4

```

Strings can also be compared using the comparison operators in Table 34. String comparison works much the same way it does when you alphabetize words or names. That is, 'a' comes before 'b', and 'b' before 'c', etc. Additionally, uppercase letters come before lowercase letters and numbers come before letters. These relationships are derived, by default, from the letters' relative positions as code points in Unicode.

```

1 a = "alan"
2 b = "grace"
3 a == b → False
4 a != b → True
5 a < b → True
6 a > b → False
7 a > "Alan" → True
8 a > "123" → True

```

The syntax of most of the operations we have surveyed up to this point look casually like operators or functions you are familiar with from mathematics. But many values in Python also have special functionality that can be invoked

through methods. Calling a method is similar to calling a function except that we use a period to suffix the method we'd like to call on the object.

#### method call



In this example we call the `capitalize()` method on a string:

```

1 a = "houston"
2 a.capitalize() → "Houston"
  
```

Methods and functions are closely related but they are not interchangeable. A function isn't normally invoked as a method and a method isn't normally invoked as a function.

```

1 a = "houston"
2 a.len() → Error
3 capitalize(a) → Error
  
```

Strings have a rich set of methods listed in the Python Documentation section on the [string module](#). Some of the most common string methods are summarized in Table 39.

Table 39: *String Methods*

Operation	Description
<code>x.capitalize()</code>	Returns a capitalized string from <code>x</code>
<code>x.join(seq)</code>	Returns a string formed by joining <code>seq</code> with <code>x</code> as the separator
<code>x.lower()</code>	Returns a lower cased string
<code>x.strip()</code>	Returns a string with left and right white space removed
<code>x.split(s)</code>	Returns a list of substrings obtained by splitting <code>x</code> on every occurrence of <code>s</code>
<code>x.upper()</code>	Returns an upper cased string

### 3.5 Lists and Dictionaries

Python uses lists and dictionaries as containers to hold other values. Lists are used to hold a sequence of values. Lists can be formed using hard braces with comma separated values.

```
1 shapes = ["Rock", "Paper", "Scissors"]
```

When hard braces are suffixed to a list variable we can access elements from the list. Like strings, we count from zero.

```
1 shapes[0] → 'Rock'
```

Lists (and list-like objects) are important building blocks in Python and provide a powerful mechanism for organizing information. Compare this rock-paper-scissor player to the one earlier in the chapter. It has half the code and the connection between shapes and random selection is more explicit.

```
1 import random
2 shapes = ["Rock", "Paper", "Scissors"]
3 print(random.choice(shapes) + "!")
```

It's also easier to extend this version to include new shapes.

```
1 import random
2 shapes = ["Rock", "Paper", "Scissors",
3          "Lizard", "Spock"]
4 print(random.choice(shapes) + "!")
```

Dictionaries are used to map one object (called the key) to another object (called the value) much like a real dictionary maps a word to a definition. Dictionaries are formed from curly braces and key value pairs.

```
1 color_frequency = {"red": 650,
2                   "green": 510,
3                   "blue": 475}
```

We access the values in a dictionary using the appropriate key.

```
1 color_frequency["red"] → 650
```

We will explore these and other collections in more depth in Chapter 6.



### 3.6 None

The final type we will cover in this Chapter is the `None` type, which is usually used to indicate that no value is present<sup>6</sup>.

#### none literal



We have already encountered one function that returns `None`:

```
1 print(print())
```

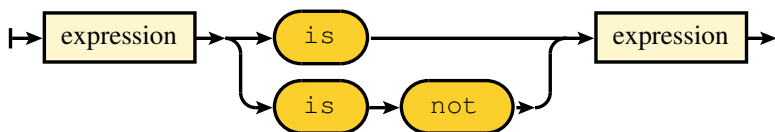
```
1 Output:
```

```
2
```

```
3 None
```

While `print()` displays a value as output, it doesn't actually return anything. When we use `None` explicitly, we are usually concerned about the identity of an object (which we will explore in more detail in Section 9.7).

#### identity expression



One common practice is to set variables which have not yet been initialized and those that were unsuccessfully initialized to `None`:

```
1 user = get_user()
```

```
2
```

```
3 if user is None:
```

---

<sup>6</sup> `None` is very similar to `NIL` or `NULL` used in other languages but unlike most `NIL` or `NULL` values Python's `None` is a true object with its own type.

```

4     print("Sorry, who are you?")
5     else:
6     print("Welcome back!")

```

### 3.7 Comparing Types

Sometimes we want to convert one type of value into another type of value. For example, perhaps we asked the user to enter an integer with the `input()` function. If you worked on problem 2-14 then you probably learned that `input()`'s return type is always a `str`.

But if you try to compare a string and an integer you get an error:

```

1 "70" >= 65 → TypeError

```

To convert a string into an integer, we use the `int()` function as illustrated in the next example:

```

1 age = input("Enter your age in years: ")
2 age = int(age)
3 if age >= 65:
4     print("The movie ticket costs $6.00")
5 else:
6     print("The movie ticket costs $8.00")

```

Python has a number of methods that can be used for doing conversions between `ints`, `floats`, and `strs` but some of the most common are summarized in Table 310.

Table 310: *Conversion Functions*

Operation	Description
<code>str(x)</code>	Converts <code>x</code> to a human readable string representation <sup>7</sup>
<code>int(x)</code>	Converts a string or number, <code>x</code> , to an <code>int</code> .
<code>float(x)</code>	Converts a string or number, <code>x</code> , to a <code>float</code> .

7. Converting numbers to strings is a particularly common application but `str()` doesn't really give us much control over this conversion. In Section 4.7 we discuss the `format()` method which provides much finer control over how numbers are represented as strings.

Python also lets us look at the type directly with the built-in `type()` function:

```
1 type(True) → <class 'bool'>
2 type(42) → <class 'int'>
3 type(3.14) → <class 'float'>
4 type("Flagstaff") → <class 'str'>
```

In situations where we don't know what type is associated with a variable, we can use `type()` to make decisions:

```
1 if type(value) == str:
2     create_string_node(value)
3 elif type(value) == int:
4     create_int_node(value)
5 else:
6     error()
```

## Glossary

**accuracy** The correctness of a value.

**arbitrary-precision arithmetic** See `bignum`.

**assignment** Associating a value with variable.

**bignum** A language feature that allows numbers to be represented with arbitrary precision. Also known as arbitrary-precision arithmetic.

**bool** Python's Boolean data type, which can represent `True` or `False` values.

**Boolean** A binary value representing either true or false.

**clock arithmetic** Another name for the modulus operation.

**comment** A human readable source code annotation that is not evaluated.

**comparison operators** Operators which compare their operands and yield a Boolean result.

**conversion** A transformation from one type to another that fundamentally changes the underlying representation and type.

**data type** A classification used to distinguish different kinds of values.

**data structure** A scheme for organizing and storing related data.

**dynamic typing** A language property where a variable's type can change over time.

**escape sequence** A special sequence that has special meaning within a string literal.

**float** A numeric data type, which represents real numbers with limited representation.

**identifier** A named language entity.

**if statement** A conditional structure that can change the flow of execution in a program.

**int** A numeric data type for representing integers.

**integer division** Integer division is equivalent to the result of real number division followed by the mathematical floor. The resulting number is an `int`.

**keyword** An identifier defined and reserved by a programming language.

**literal** A primitive expression where the textual representation directly represents the value.

**logical operators** Operators that work on true and false values using logical conjunction, disjunction, and negation.

**precision** The tightness or tolerance of a specification.

**static typing** A language property where a variable's type can not time.

**string** A data type used for representing text.

**truth table** A table of Boolean variables, logical expressions, and their corresponding possible values.

**value** A data item that can be manipulated within a computation.

**variable** A named location in memory where values can be stored.

### Reading Questions

- 3.1. How do programming languages distinguish different kinds of information?
- 3.2. How is the assignment operation different from other Python expressions?
- 3.3. What is the relationship between variables and values?
- 3.4. Which is not a literal value?

```
1 22.0
2 red
3 "blue"
4 False
```

- 3.5. Give an example where a Python variables takes on different values over time.
- 3.6. Give an example where a Python variables takes on different types over time.
- 3.7. What characters can be used to start an identifier?
- 3.8. Which is not a valid identifier?

```
1 two_pi
2 2pi
3 TwoPi
4 pi2
```

- 3.9. What is a keyword?
- 3.10. What properties makes a variable name “good”?
- 3.11. What is a truth table?
- 3.12. What kind of statement allows conditional actions?
- 3.13. What is the purpose of the `elif` and `else` clauses?
- 3.14. How big can an int be?
- 3.15. Are floats inaccurate?
- 3.16. What is the relationship between integer division and the modulo operator?
- 3.17. Why does Python have string escape sequences?
- 3.18. Do different kinds of quotes create different kinds of strings?
- 3.19. How is the syntax for method calls and function calls different?
- 3.20. What Python types did we discuss in this chapter?

**Exercises****3-1. Roman numerals**

Write a program that prompts the user to enter a number between 1 and 5 (inclusive). Your program should then print the corresponding Roman numeral (i.e., I, II, III, IV, or V) or print an error message for invalid input.

Sample run:

```
1 Enter a number between 1 and 5: 4↵
2
3 IV
```

**3-2. Complementary colors**

Complementary colors are colors that when combined under a particular color model produce white or black. Write a program that prompts the user for a color. Your program should then print the complementary color or print an error message for invalid input.

- a) red ↔ green
- b) yellow ↔ violet
- c) blue ↔ orange

**3-3. Chess values I**

Chess pieces are commonly assigned values to provide a rough guide to the state of play. Write a program that asks the user to name a chess piece and then prints the corresponding value. If the input does not correspond to a chess piece your program should print an appropriate error message.

- a) pawn → 1
- b) knight → 3
- c) bishop → 3
- d) rook → 5
- e) queen → 9

**3-4. Odd or even?**

Write a program that prompts the user for an integer and then tells the user if the number was odd or even.

Sample run:

```
1 Enter an integer to check if even or odd: 7↵
2
3 The number 7 is odd.
```

**3-5. Simple scoring**

Write a program that prompts the user for a grade and then tells the user if the grade was an A (90-100), B (80-90 exclusive), C (70-80 exclusive), D (60-70 exclusive) or F (below 60).

Sample run:

```
1 Please enter a grade: 76↵
2
3 76 gives you a C.
```

**3-6. Miles to kilometers**

Write a program that prompts the user for a distance in miles and then prints the distance in kilometers (1 mile = 1.60934 kilometers.)

Sample run:

```
1 Enter a distance in miles: 26.7↵
2
3 26.7 miles converts to 42.9695 kilometers.
```

**3-7. Fahrenheit to celsius**

Write a program that prompts the user for a temperature in fahrenheit and then prints the temperature in Celsius ( $(F - 32) * 5/9 = C$ ).

Sample run:

```
1 Enter a temperature in fahrenheit: 98.6↵
2
3 37 Celcius
```

**3-8. Area of a sphere**

Write a program that calculates and displays the surface area of a sphere given the sphere's radius. The formula for computing the area of a sphere is  $A = 4\pi r^2$ .

Sample run:

```
1 Enter radius: 5↵
2
3 The area of the sphere is:
4 314.1592653589793
```

**3-9. Area of a cube**

Write a program that calculates and displays the surface area of a cube given an edge length.

The formula for computing the area of a cube is  $A = 6e^2$ , where  $e$  is the length of an edge.

Sample run:

```
1 Enter edge length: 3↵
2
3 The area of the cube is:
4 54
```

### 3-10. Time since midnight

Write a program that takes the hours, minutes, and seconds from midnight and displays the total number of seconds.

Sample run:

```
1 Enter hours: 4↵
2 Enter minutes: 30↵
3 Enter seconds: 46↵
4
5 Total seconds from midnight:
6 16246
```

### 3-11. A matter of time

Write a program that prompts the user to enter some number of seconds ( $x$ ) since midnight. Then display the time using the sentence, “ $x$  corresponds to  $h$  hours,  $m$  minutes, and  $s$  seconds,” where  $x$ ,  $h$ ,  $m$ , and  $s$  are numbers such that  $m < 60$  and  $s < 60$ .

Sample run:

```
1 Enter seconds since midnight: 30601↵
2
3 8 hours , 30 minutes , 1 second(s)
```

### 3-12. Unit pricing

Supermarkets often display prices per unit to make it easier for shoppers to compare prices of similar items that may come in different sizes or amounts. For example, a four-ounce item that costs 80 cents has unit price of 20 cents per ounce. Write a program that takes the weight of the item as its first input and the price of the item as its second input. The program should print the unit price of the item.

Sample run:

```
1 Enter product weight in ounces: 17 ↵
2 Enter product price: 3.48↵
3
4 Product costs $.20 per ounce.
```

### 3-13. Triangular

Suppose you had three sticks of length 2 inches, 3 inches, and 6 inches. You would be unable to connect the three sticks into a triangle without



cutting one of the sticks. Three sticks can make a triangle if the sum of each pair of lengths is greater than the third. Write a program that prompts the user for three numbers and displays a message indicating if the three lengths can form a triangle.

Sample run:

```
1 Enter first number: 3↵
2 Enter second number: 4↵
3 Enter third number: 5↵
4
5 Lengths 3, 4, and 5 can make a triangle.
```

### 3-14. Pig Latin translator

Pig Latin is a simple language game where words are changed according to simple rules. Write a program that prompts the user to enter a word. Your program should print the corresponding Pig Latin word based on these two rules:

- a) If the word starts with a consonant, move it to the end of the word and add “ay” (e.g., happy - appyhay)
- b) If the word starts with a vowel, just add “yay” to the end of the word.

### 3-15. Phone number fix

A company keeps phone numbers in two different formats but wants to standardize on one format. Write a program that prompts the user for a phone number - it should accept both styles below as input but it should always print the number using the second style. Print an error if the input conforms to neither style (handling unexpected input properly may be harder than you think!). The two styles you should consider are:

- a) (123)456-7890
- b) 123-456-7890

### 3-16. Approximation of $\pi$

Write a program that calculates and displays an approximation of  $\pi$  using the formula  $\pi \approx 4 * (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11})$ .

Sample run:

```
1 2.9760461760461765
```

### 3-17. Length of a vector

The formula for the magnitude (length) of a 2D vector is  $|a| = \sqrt{x^2 + y^2}$ . Write a program that takes user input to store the components of the vector  $\vec{a} = [x, y]$  and then displays the vector and its magnitude.

Sample run:

```
1 Enter x: 7↵
2 Enter y: 20↵
3
4 The length of the vector [7, 20] is:
5 21.18962010041709
```

Can you extend your program to work with 3D vectors?

### 3-18. Tip calculator

Write a program that calculates the tip you should give your host. Have the user input the meal price and the level of service. Some common tipping rates used in America include: poor service 15%, good service 20%, excellent service 25%.

Sample run:

```
1 Enter your meal price: 42.65↵
2 How was your service? excellent↵
3
4 The total price with tip:
5 51.18
```

### 3-19. Rock-paper-scissors I

Write a program that prompts the user to type “rock,” “paper,” or “scissors.” Then have the computer choose one of the same shapes (see the code we developed in this chapter). Compare the shape chosen by the computer to the one chosen by the user and using the rules of **rock-paper-scissors** declare a winner.

Sample run:

```
1 rock, paper, or scissors? rock↵
2
3 You chose rock.
4 The computer chose scissors.
5 You win!
```

### 3-20. Rock-paper-scissors II

Write a program that **cheats** at Rock-paper-scissors! It should work similar to Problem **3-19** but instead of randomly selecting a choice for the computer, make the choice that wins!

Sample run:

```
1 rock, paper, or scissors? rock↵
2
3 You chose rock.
4 The computer chose paper.
5 The computer wins!
```

## Chapter Notes

This chapter introduces variables and values as important programming building blocks. Every value in Python has a type which defines a value's internal representation and provides operations for that type. This chapter surveys some of Python's most commonly used types:

1. `bool`, which is used for representing `True` and `False` values,
2. `int`, which is used for representing positive and negative integers (e.g., -2, -1, 0, 1, 2),
3. `float`, which is used for representing positive and negative real numbers (e.g., -2.7, 0.005, 3.14),
4. `str`, which is used for representing text (e.g., `"hello_world"`, `"aaga"`),
5. `list`, which is used to represent an ordered sequence of values, (e.g., `[11, 42, 13]`), and
6. `dict`, which is used to represent a mapping between keys and values (e.g., `{"name": "dave", "password": "qwerty"}`)

We also surveyed many of the operators, functions, and methods which are commonly used with each of these types. You will need to become very comfortable using these basic operations and using types and values as a means to represent properties and relationships in simple problems.

- [31] George Boole. **An Investigation of the Laws of Thought**. Dover Publications, Incorporated, 1958. [isbn: 0486600289](#).
- [32] David Goldberg. What every computer scientist should know about floating-point arithmetic. **ACM Computing Surveys**, 23(1):5–48, March 1991. [doi: 10.1145/103162.103163](#).
- [33] Donald E. Knuth. **Art of Computer Programming, Volume 2: Seminumerical Algorithms**, pages 265–280. Addison-Wesley Professional, Reading, Mass, third edition, 1997. [isbn: 9780201896848](#).
- [34] J. Korpela. **Unicode Explained**. O'Reilly Media, 2006. [isbn: 9780596101213](#).



## 4

# Compound Expressions

*The purpose of computing is insight, not numbers.*

*Richard Hamming*

```
1 overview = ["Function_Composition",
2             "Method_Chaining",
3             "Operator_Precedence",
4             "Short-Circuit_Evaluation",
5             "Truthiness",
6             "Compound_Assignment_Operators",
7             "String_Formatting",
8             "Glossary",
9             "Reading_Questions",
10            "Exercises",
11            "Chapter_Notes"]
```

In the last chapter we discussed core operations for several built-in Python types. You may have noticed that different operators had different syntax. So far we have seen four different expression syntaxes:

1. **infix operations**, which consists of an operator between two operands (e.g., `x + y`, `x * y`),
2. **prefix operations**, which consists of an operator before an operand (e.g., `-x`, `not x`),

3. **function calls**, which consists of a function name followed by a comma separated list of arguments (e.g., `abs(x)`, `len(x)`), and
4. **method calls**, which consists of an object followed by a dot, a method name, and a comma separated list of parameters (e.g., `x.capitalize()`)

In this chapter we build on these simple expressions to form more complicated expressions and discuss many of the nuances of expression evaluation.

#### 4.1 Function Composition

**Function composition** is a way of combining functions such that the result of one function is passed as the argument of another function[35]. For example, the composition of two functions  $f$  and  $g$  is denoted  $f(g(x))$ . Here  $x$  is the argument of  $g$ , and the result of  $g$  is passed as the argument of  $f$ . Syntactically, the operand or parameter of an expression may be another expression.

Lets consider a concrete example where we convert a number with an arbitrary base to base 10. We might write the program such that each function call is relatively flat by using temporary variables to store the result of each function:

```

1 number_str = input("Enter a number: ")
2 base_str = input("Enter its base: ")
3
4 base_int = int(base_string)
5 number_int = int(number, base_int)
6 print(number_int)

```

```

1 Output:
2 Enter a number: D5↵
3 Enter a base: 16↵
4 213

```

By using function composition, we can combine the last three lines of the program into one, which avoids temporary variables and makes the relationships between functions explicit:

```

1 number = input("Enter a number: ")
2 base = input("Enter its base: ")
3
4 print(int(number, int(base)))

```

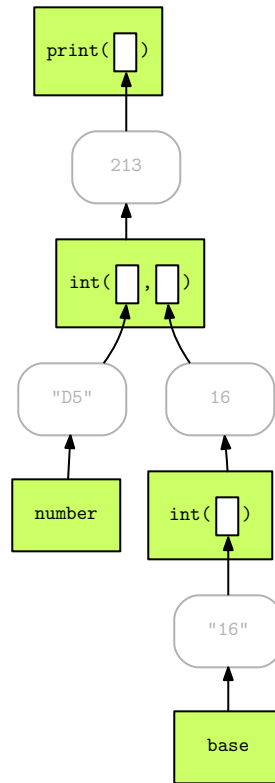


Figure 41: The green boxes represent an expression evaluation and the rounded white boxes represent the result of each evaluation. Functional composition occurs as one function's result becomes another function's argument.

When functions are composed like this we evaluate the inner-most expression first. The result becomes the argument to the enclosing expression, and so on. We can visualize this process in something called an **expression tree**. See Figure 41.

First, Python will evaluate `int(base)` first to convert the `str`, `base`, to an `int`. Second, Python will evaluate the enclosing `int()` with `number` and `base` as arguments. And finally, Python will evaluate the `print()`.

Evaluating the full expression anchored at the top requires that we evaluate its branching subexpressions. When viewed as a tree, we can think of evaluation as starting at the leaves (nodes with no branches) and values spread upward combining and changing with each expression.

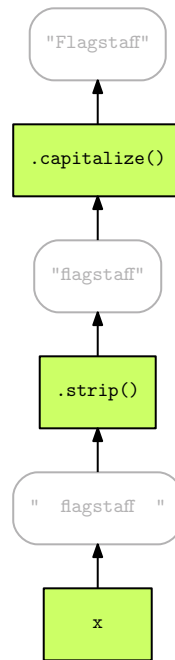


Figure 42: Method chaining occurs when a method is invoked on the result of another method.

## 4.2 Method Chaining

Methods have an analogous mechanism for composition called method chaining. In method chaining, when a method returns a value, this value becomes the object for the next method call. Lets consider an example where we strip the white space from a string and then capitalize it:

```

1 x = "   flagstaff   "
2
3 # Remove the whitespace
4 x = x.strip()
5
6 # Capitalize
7 x = x.capitalize()
8
9 print(x)

```

The same thing can be accomplished more succinctly using method chaining:



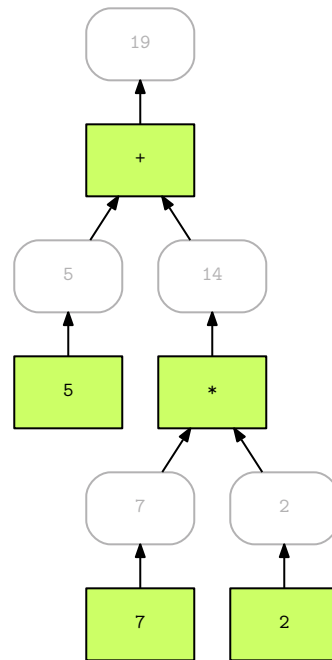


Figure 43: Operator precedence dictates how infix expressions are combined to yield a result.

```

1 x = "   flagstaff   "
2
3 print(x.strip().capitalize())

```

### 4.3 Operator Precedence

Infix operations can also be chained together in a way that you are probably already familiar:

```

1 y = 5 + 2 * 7

```

How is `y` evaluated in the last example? Do you add or multiply first? You are probably already familiar with the concept of **operator precedence** (also called order of operations) where some operations have higher precedence in evaluation than others. Multiplication, for example, has higher precedence than addition meaning that `y` takes the value 19 in the last example.

Figure 43 illustrates how we might design an expression trees for the last

expression. Notice that each operator node has two operands. When multiple operators are connected together, a binary tree is formed. Python relies on an internal representation similar to an expression tree to parse infix expressions into a data structure that preserves precedence.

Python extends the operator precedence you are already familiar with to include comparison and logical operators as ranked in Table 41<sup>1</sup>. Operators with high precedence are the most binding and those with the lowest precedence are the least binding.

Table 41: Operator Precedence

Precedence	Operator Family
High	<code>()</code> , <code>[]</code> , <code>{}</code>
	<code>x[]</code> , <code>x()</code> , <code>x.attribute</code>
	<code>**</code>
	<code>+x</code> , <code>-x</code>
	<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>
	<code>+</code> , <code>-</code>
	<code>in</code> , <code>not in</code> , <code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code> , <code>!=</code> , <code>==</code>
	<code>not x</code>
	<code>and</code>
	<code>or</code>
Low	<code>lambda</code>

In order to understand what an expression does, it's sometimes useful to fully parenthesize the expression. To fully parenthesize an expression we look for the highest precedence operations in an expression and group them together with parenthesis. Operators of equal precedence are grouped from left to right. Several examples of operator precedence are provided in Table 42.

Table 42: Operator Precedence Examples

Example	Fully Parenthesized Example
---------	-----------------------------

1. Operations not discussed in this text have been omitted.

Example	Fully Parenthesized Example
$10 - 4 + 2$	$(10 - 4) + 2$
$10 - 4 * 2$	$10 - (4 * 2)$
$p + q * r + s$	$(p + (q * r)) + s$
$p + q * r + s / t$	$(p + (q * r)) + (s / t)$
$p \text{ and } q \text{ or } r \text{ and } s$	$(p \text{ and } q) \text{ or } (r \text{ and } s)$
$p \text{ and } q \text{ or } r \text{ and } s \text{ or } t \text{ and } u$	$((p \text{ and } q) \text{ or } (r \text{ and } s)) \text{ or } (t \text{ and } u)$

If you aren't familiar with Python's precedence rules, you may be tempted to add "clarifying" parenthesis when you aren't confident about the precedence but in most cases adding extraneous parenthesis only adds more visual clutter.

#### 4.4 Short-Circuit Evaluation

In working with Booleans and logical operators you may have noticed an interesting property: sometimes the result can be determined by looking just at the first operand.

Many programming languages, including Python, will not evaluate the second operand of a logical operator when the first operand determines the result. When the second operand isn't evaluated we call this a **short-circuit**. Both the conjunctive (`and`) and disjunctive (`or`) operators may cause a short-circuit in evaluation.

If the first operand to `and` is `True` the second operand must be evaluated in order to return a result (as shown left in Figure 44). But if the first operand to `and` is `False` then the second operand does not contribute to the result and Python short-circuits (as shown right in Figure 44). Python will not evaluate any expressions in the expression tree below the short-circuit (colored red in the Figure).

A short-circuit may also occur when working with `or`. If the first operand is `False` then the second operand must be evaluated. But if the first operand is `True` then the second operand is short-circuited (as shown in Figure 45).

Short-circuits allow subexpressions to be evaluated conditionally enabling programs to branch without an explicit `if` statement.

In this example a short-circuit occurs if `is_awesome()` is `True`:

```
1 is_awesome() or exit()
```

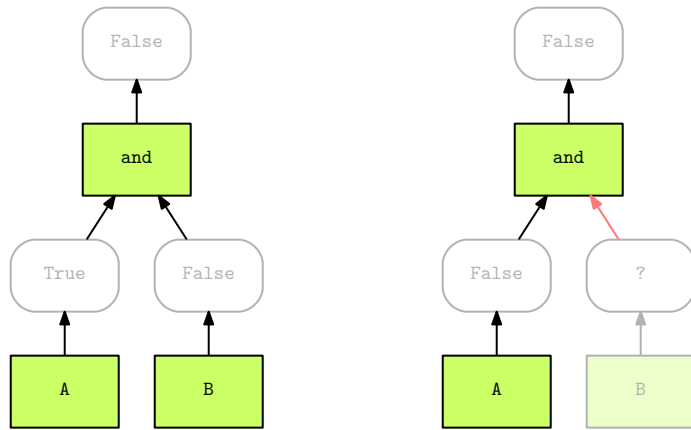


Figure 44: *and* short-circuits (colored red) if the first operand is *False*.

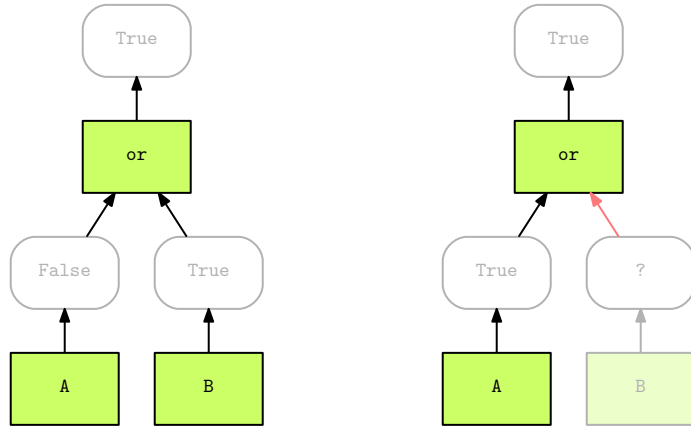


Figure 45: *or* short-circuits (colored red) if the first operand is *True*.

The result is that the program will exit if `is_awesome()` is `False`. The equivalent code without a short-circuit might read:

```
1 if not is_awesome():
2     exit()
```

Short-circuit evaluation can be a double edged sword: short-circuit expressions can be short and uncluttered but they can also be overly complicated and hard to understand.

#### 4.5 Truthiness

Every value in Python has a property called **truthiness** which determines if the value behaves like a Boolean `True` or a Boolean `False` when used as a test. In evaluating a logical expression, the following values act like `False`:

- `False`
- `0 (int)`
- `0.0 (float)`
- `None`
- Empty containers (strings, list, dictionaries, etc.)

Almost any other value acts like `True`.

In this coin flipping example we use the truthiness of the `int` returned by `randint()` to decide if the flip was heads or tails without explicitly doing a comparison:

```
1 import random
2
3 if random.randint(0, 1):
4     print("Heads!")
5 else:
6     print("Tails!")
```

When truthy values are combined with logical operators the result of an expression is determined by the last value which determined the truthiness of the expression. Thus, understanding short-circuit evaluation is critical to understanding logical expressions made up of values which may not strictly be of type `bool`.

The result of this expression is 7 because its truthy value is used over zero which acts like a False:

```
1 0 or 7 → 7
```

The result of this expression is 5 because its truthy value short-circuits the 7:

```
1 5 or 7 → 5
```

Similarly, `and` short-circuits on a falsey value:

```
1 0 and 7 → 0
```

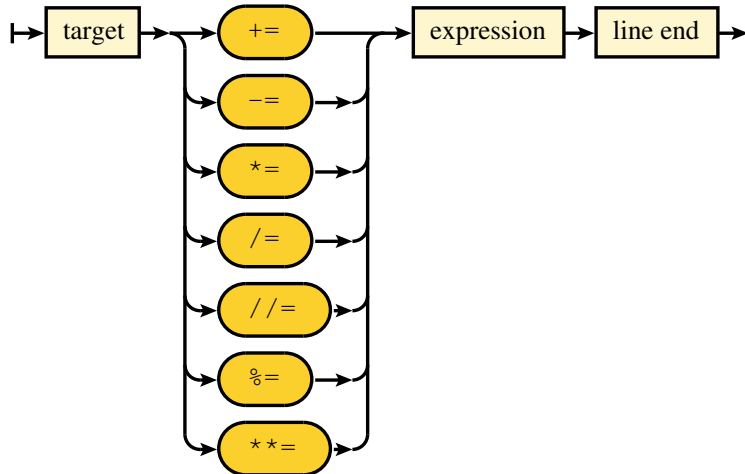
Values of different types follow the same rules and different types can be freely mixed in an expression:

```
1 "A" or "B" → "A"  
2 "A" and "B" → "B"  
3 0 or "C" → "C"  
4 None and "C" → None  
5 [] or ["Flagstaff"] → ["Flagstaff"]
```

While truthiness is a common concept in dynamically typed languages it's less common in statically typed languages where logical operators and tests may expect only one type of value.

#### 4.6 Augmented Assignment Operators

Most arithmetic infix operations can be combined with assignment to create compound assignment operators.

**augmented assignment statement**

Augmented assignment operators can be used to make code more readable. We will often want to do simple operations like incrementing a variable by one. Once you are accustomed to the syntax it can be easier to read `x += 1` as “add one to x” rather than `x = x + 1`.

In reading expressions with long variable names we sometimes have to ask ourselves if the variable on the right is the same as the one on the left. Here we have a typo you could easily miss:

```
1 number_of_widgets = number_widgets + 1
```

The same mistake is more difficult to make (or at least there are fewer opportunities for error) with the augmented assignment:

```
1 number_of_widgets += 1
```

The same syntax also works with non-numeric types:

```
1 greeting = "hello"
2 greeting += " world"
3 greeting → "hello world"
```

Augmented assignment operators combine an expression and an assignment but they are considered assignment operations and as such can not be used as expressions.

## 4.7 String Formatting

Often times we need careful control over how Python displays data. We can get this control through a process called **formatting** where Python data types are converted into a human readable form. Python has its own mini-language just for doing this.

Format strings contain a special markup for **replacement fields** which are designated by curly braces `{}`. Anything that is not contained in braces is considered literal text and anything inside the braces is replaced with values passed to the method. A brace can be included in the literal text by repeating it twice (i.e., `{` and `}`).

There are three ways of specifying which value corresponds to each field: implicit order, explicit order, and named order.

### *Implicit Order*

With implicit ordering, the replacement fields may be blank and the order of the arguments to the format method designate the order in the string. Consider this example:

```
1 >>> "{}_{}_{}".format("z", 42, True)
2 z 42 True
```

### *Explicit Order*

With explicit ordering, the replacement fields specify specifically which value should be used:

```
1 >>> "{2}_{}_{}".format("z", 42, True)
2 True z 42
```

### *Named Order*

With named ordering, we use parameter names to link the replacement field to the value:

```
1 >>> "{c}_{}_{}".format(a="z", b=42, c=True)
2 True 42 z
```



*Format Operations*

Finally, we can control how each variable is displayed with format codes. Format codes appear inside the replacement code after the variable specifier. Some common format codes are listed in Table 43.

Table 43: *A Few Common Format Codes*

Operation	Description
:f	Display as a real number
:d	Display as decimal
:x	Display as hex
:o	Display as octal
:b	Display as binary
:m	Format the number such that it has width m.
:m.n	Format the number such that it has m total width and n digits after the decimal point

Format codes allow us to change how we display values. This can be used to make our output more readable, create certain textual effects (like alignment), or meet formatting requirements for interoperability with other programs.

```
1 # print the number so it takes
2 # up four spaces:
3 print("{:4}".format(42))
4
5 # print the number in binary
6 print("{:b}".format(42))
7
8 # print the number with 4
9 # decimal places
10 print("{:.4}".format(42))
```

The format codes we have presented here are a very small subset of the complete formatting language. A more complete list of format codes and a description of the replacement field mini-language is given in the Python Documentation section on [Format String Syntax](#).

## Glossary

- string formatting** Arranging and setting how values are displayed as strings.
- function composition** When functions are combined such that the result of one function becomes the argument to another function.
- infix** Syntactic form where operators appear between operands.
- last-value evaluation** The principle that a logical expression yields the last evaluation as its result.
- method chaining** When the result of one method is directly used with another method.
- operator precedence** When infix operations are chained together the evaluation order is determined by the relative precedence of the operator. Higher precedence operations are evaluated first. Also known as order of operations.
- order of operations** See **Operator Precedence**.
- postfix** Syntactic form where operators appear after operands.
- prefix** Syntactic form where operators appear before operands.
- replacement fields** Specially marked up text that control how values are displayed in a format string.
- short-circuit evaluation** The principle that logical expressions will not evaluate subexpressions where the value is inconsequential.
- truthiness** The principle that values can be interpreted as being logically true or false without necessarily being the `bool` values `True` or `False`.

## Reading Questions

- 4.1. What is a return value?
- 4.2. What is a parameter?
- 4.3. In what order do we evaluate expressions that have been combined with function composition?
- 4.4. In what order do we evaluate expressions that have been combined using method chaining?
- 4.5. Can function composition and method chaining be combined? Why or why not?

- 4.6. In what order do we evaluate expressions that have been combined using infix operators?
- 4.7. If no logical operators are involved in an expression is short-circuit evaluation possible? Explain.
- 4.8. Which has higher precedence “and” or “or”?
- 4.9. Which has higher precedence logical operators or arithmetic operators?
- 4.10. Which has higher precedence comparison operators or logical operators?
- 4.11. Which values act like False with respect to truthiness?
- 4.12. If all the values in a logical expression are Boolean, does last-value evaluation happen? Explain.
- 4.13. Can compound assignment operators be used as expressions?
- 4.14. What does string formatting do?
- 4.15. If the replacement fields in a format string are empty, what order does it take its arguments?
- 4.16. What is the advantage of explicit order v. implicit order in string formatting?
- 4.17. What is the advantage of named order v. explicit order in string formatting?
- 4.18. What string format code would display a number in hex?
- 4.19. What is the advantage of string formatting over using the functions and operators that were introduced in the last chapter?
- 4.20. When we visualize an expression graphically we call it an expression tree. Why is it called a **tree**?

## Exercises

### 4-1. Distance between two points

Write a program that prompts the user for two points (as described by the four variables  $x_1$ ,  $y_1$ ,  $x_2$ , and  $y_2$ ) and displays the distance between those points. The formula for computing the distance,  $d$ , between two points is  $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ .

Sample run:

```

1 Distance Between Two Points
2 =====
3 Enter x1: 2↵
4 Enter y1: 3↵
5 Enter x2: 7↵
6 Enter y2: 12↵
7 The distance between the two points
8 is 10.295630140987.

```

#### 4-2. The midpoint formula

Write a program that prompts the user for two points (as described by the four variables  $x_1$ ,  $y_1$ ,  $x_2$ , and  $y_2$ ) and displays the midpoint. The formula for computing the midpoint is  $(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2})$ .

Sample run:

```

1 Midpoint Calculator
2 =====
3 Enter x1: 5↵
4 Enter y1: 8↵
5 Enter x2: 12↵
6 Enter y2: 16↵
7 The midpoint is 8.5, 12.0.

```

#### 4-3. Straight line equation

Write a program that prompts the user for a point  $(x, y)$  and a  $y$ -intercept and then displays the slope of the corresponding line. Recall the formula for a straight line is  $y = mx + b$ , where  $m$  is the slope and  $b$  is the  $y$ -intercept.

Sample run:

```

1 Slope Calculator
2 =====
3 Enter x: 9↵
4 Enter y: 9↵
5 Enter the y-intercept: 0↵
6 The slope is 1.0.

```

#### 4-4. Quadratic formula

Write a program that prompts the user for the coefficients of a quadratic polynomial and then prints the solution(s) to the polynomial. Beware of the square root of negative numbers when testing!

Sample run:

```

1 Quadratic Calculator
2 =====

```

```

3 ax**2 + bx + c = 0
4 Enter a: 1↵
5 Enter b: 8↵
6 Enter c: 4↵
7 The solutions are -0.5358983848622456
8 and -7.464101615137754.

```

#### 4-5. Base converter

Write a program that prompts the user for a number and its base. Then print the resulting value as a) binary, b) octal, c) decimal, and d) hexadecimal.

Sample run:

```

1 Base Converter
2 =====
3 Enter an integer: 42↵
4 Enter a base: 10↵
5 binary: 101010
6 octal: 52
7 decimal: 42
8 hex: 2A

```

#### 4-6. Character code convertor

Write a program that takes a single character as input and converts it to the decimal, hex, or binary representation of the characters' numeric value, then display the result.

Sample run:

```

1 Character Code Calculator
2 =====
3 Enter a character: h↵
4 Display as dec, hex, or bin? hex↵
5 0x68

```

#### 4-7. Wind chill

Wind Chill describes the rate of heat loss on the human body based on the combined effect of temperature ( $T$ ) and wind ( $V$ ). The National Weather Service defines the Wind Chill Index as  $T(wc) = 35.74 + 0.6215T - 35.75(V^{0.16}) + 0.4275T(V^{0.16})$ , where  $T$  and  $V$  are given in Fahrenheit and Miles Per Hour respectively. Write a program that given  $T$  and  $V$ , computes the wind chill.

Sample run:

```

1 Wind Chill Calculator
2 =====

```

```

3 Enter temperature (F): 40↵
4 Enter wind speed (mph): 30↵
5 The wind chill is 28.462045104487707.

```

#### 4-8. Body Mass Index

Body Mass Index (BMI) is a number calculated from a person's weight and height. BMI does not measure fat directly, but research has shown that BMI correlates to direct measures of body fat. The BMI formula is weight in pounds divided by height in inches squared and multiplied by 703.

Sample run:

```

1 Body Mass Index Calculator
2 =====
3 Enter weight (pounds): 145↵
4 Enter height (inches): 70↵
5 Your BMI is 20.803061224489795.

```

#### 4-9. Making change

Write a program that prompts the user for the amount of change that should be dispensed to a customer (in cents). Then print the number of quarters, dimes, nickels and pennies needed to give correct change. Your program should dispense as few coins as possible.

Sample run:

```

1 Change Calculator
2 =====
3 Enter amount (cents): 233↵
4 Dollars: 2
5 Quarters: 1
6 Dimes: 0
7 Nickels: 1
8 Pennies: 3

```

#### 4-10. Leap years

A solar year is not exactly 365 days long. In order to adjust for this, a leap year is added according to these rules of increasing precedence:

Year	Result	Examples
Not divisible by 4	Not a leap year	2006, 2007, 2009
Divisible by 4	Leap year	2008, 2012, 2016
Divisible by 100	Not a leap year	1800, 1900, 2100
Divisible by 400	Leap year	2000, 2400

Write a program that prompts the user to enter a year and then determines if the year is a leap year.

Sample run:

```

1 Leap Year Calculator
2 =====
3 Enter a year: 1980↵
4 1980 is a leap year.

```

#### 4-11. Writing numbers in English

The number 42 is written FORTY TWO. Write a program that takes a two digit integer as input and then prints the written English form of the number. Hint: You can get pretty far using modulo and if statements in this problem.

Sample run:

```

1 English Number Translator
2 =====
3 Enter an integer: 22↵
4 TWENTY TWO

```

#### 4-12. Seasons

Write a program that prompts the user for a month and a day and then prints the corresponding season.

Sample run:

```

1 Season Calculator
2 =====
3 Enter a month: 12↵
4 Enter a day: 22↵
5 The season is Winter.

```

#### 4-13. Printing dates I

Write a program that prompts the user for an eight digit integer and then prints a long formatted date. You should be able to manage this without using Python's `date` module.

Sample run:

```

1 Date Formatter
2 =====
3 Enter a date (yyyymmdd): 19480222↵
4 February 2, 1948

```

#### 4-14. Printing dates II

Extend Problem 4-13 such that it displays an error message when a date is invalid. Don't forget about leap years (see Problem 4-10)!

**4-15. Compound interest calculator**

Write a program that calculates compounded interest. The formula for compound interest is:

$$A = P\left(1 + \frac{r}{n}\right)^{nt}$$

A = amount of money accumulated after n years, including interest

P = principal amount (the initial amount you borrow or deposit)

r = annual rate of interest (as a decimal)

n = number of times the interest is compounded per year

t = number of years the amount is deposited or borrowed for

**4-16. ISBN-10 checksum**

ISBN-10 numbers have 9 digits with a 1 digit checksum. Assuming the digits are "abcdefghi-j" where j is the check digit. Then the check digit is computed by the formula,  $j = ([a b c d e f g h i] * [1 2 3 4 5 6 7 8 9]) \bmod 11$ . Write a program that checks if an ISBN-10 number is valid. If it is invalid it should display the expected checksum.

Sample run:

```

1 ISBN-10 Validator
2 =====
3 Enter the ISBN-10 code: 8175257661↵
4 Incorrect. The last digit should be 0.
```

**4-17. Exponential decay**

Exponential decay is the decrease in a quantity N according to the law  $N(t) = N_0e^{-\lambda t}$  for a parameter t, decay constant  $\lambda$ , and initial value  $N(0) = N_0$ . Write a program that takes in the half-life of a substance in days, the initial amount in grams, and a number of days. Your program needs to calculate the decay constant  $\lambda$  and return how much is left of the initial amount of the substance after x amount of days.

Sample run (the decay rate of plutonium):

```

1 Exponential Decay Calculator
2 =====
3 Decay rate (in days): 238↵
4 Initial amount (in grams): 100↵
5 Days: 30 ↵
6 The substance has a decay constant of -0.002912
7 After 30 days there is 91.6337 grams remaining.
```

**4-18. Age group**

Write a program that takes user input for an age and displays the age group of the person. Age groups can be generically classified as: infant 0-1 years, toddler 1-2 years, child 2-13 years, teenager 13-19, young adult 19-25, adult 25-60, senior citizen 60+.



Sample run:

```

1 Age Group
2 =====
3 What is your age? 35↵
4 You are an adult.

```

#### 4-19. Run for it

The average time for a male marathon runner to complete a full marathon (42.195 kilometers) is 4 hrs, 14 minutes, and 23 seconds according to [marathonguide.com](http://marathonguide.com). Write a program that displays how many miles per hour you have to run to match the race time entered.

Sample run:

```

1 Marathon Calculator
2 =====
3 Enter hours? 4↵
4 Enter minutes? 14↵
5 Enter seconds? 23↵
6 You must run 6.184 mph.

```

#### 4-20. Amortization payment calculator

Usually, whether you can afford a loan depends on whether you can afford the monthly payment. The following amortization formula calculates the payment amount per period:

$$A = P \frac{r(1+r)^n}{(1+r)^n - 1}$$

A = payment Amount per period

P = initial Principal (loan amount)

r = interest rate per period

ex: 7.5% per year / 12 months = 0.625% per period

n = total number of payments or periods

Write a program that take user input for the loan amount 'P', the interest rate 'r', and the length of the loan in months 'n', and calculates the monthly payment.

## Chapter Notes

In this chapter we discussed several syntactic techniques for building richer more powerful expressions:

- function composition, where functions use the return values from other functions as their arguments,

- method chaining, where the return value from one method becomes the target of the next method,
- operator precedence, where the order that operations are performed is based on precedence rules,
- short-circuit evaluation, where the evaluation of an expression is effected by logical consequences,
- compound assignment operations, which combine multiple operations, and
- string formatting, which uses its own mini-language to create strings from existing values.

Python supports an alternate mechanism for formatting strings based on format codes from the C language (see the Python Documentation section on [printf Style String Formatting](#)) but it is generally discouraged in favor of the `format()` method described in this chapter. A third, more limited, mechanism for formatting strings is using template strings (see the Python Documentation section on [Template Strings](#)). The use case for template strings is slightly different - using strings as templates for things like email messages, web pages, or other longer pieces of text that need robust variable substitution. For this particular use case you might also consider several third party libraries, which provide more powerful template engines (see [Django Templates](#), [Jinja](#), [Mako](#), and [Mustache](#)).

[35] Eric W. Weisstein. Composition. **From MathWorld – A Wolfram Web Resource.** [link](#).

## 5

# Functions

*An algorithm must be seen to be believed.*

*Donald Knuth*

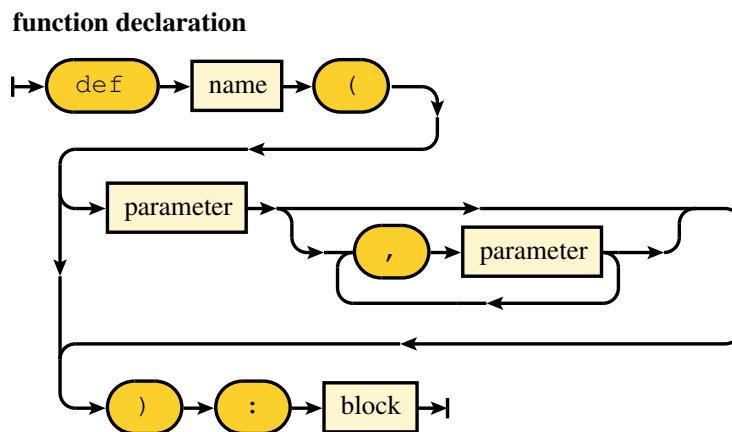
```
1 overview = ["Defining Functions",
2             "Parameters",
3             "Return Values",
4             "Docstrings",
5             "Scope",
6             "Code Repetition",
7             "Top-Down Design",
8             "Recursion",
9             "Functions as Values",
10            "Glossary",
11            "Reading Questions",
12            "Exercises",
13            "Chapter Notes"]
```

Functions provide a means of organizing reusable code that is intended to perform a single distinct task. In previous chapters we utilized Python's many built-in functions but in this chapter we will learn to create our own functions and discuss related methods of abstraction.

## 5.1 Defining Functions

Functions begin with the keyword `def`, which is then followed by the function name. This is then followed by parenthesis that enclose a list of zero or more parameters. **Parameters** are special variables that name the data, or **arguments**, that are called with a function.

The body of the function begins after a colon (`:`) and is indented. The first line of the body may optionally be a string that describes the function (called a **docstring**). The remainder of the body defines the action the function should take when called.



Functions end when the last line of the function is reached or when a special return statement explicitly exits the function.

## 5.2 Parameters

When a function is called we can think of the parameters as being copied from the call site. In this example we have defined a function to calculate the percentage of adenine bases found in a DNA fragment:

```

1 def adenine_percentage(dna):
2     return 100.0 * dna.count('A') / len(dna)
3
4 adenine_percentage('AGCATGA') → 42.857142857142854

```

When `adenine_percentage()` is evaluated, the argument `'AGCATGA'` is passed into the parameter `dna` by copying a reference to the value in a process known as **call-by-object**<sup>1</sup>.

Since a reference is copied and not the actual value, any changes to the value will be reflected in other references to the same object. We can make use of this fact to create a function that pops a value off the beginning of a list instead of the end (i.e., `list.pop()`):

```
1 def popleft(lst):
2     element = lst[0]
3     del lst[0]
4     return element
5
6 states = ['solid', 'liquid', 'gas', 'plasma']
7 popleft(states) → 'solid'
8 states → ['liquid', 'gas', 'plasma']
```

In the last example changes to the `lst` variable in `popleft()` are reflected in the `states` variable due to call-by-object<sup>2</sup>.

There is, however, an important difference between changing an object and changing a variable. Consider this alternate implementation of `popleft()`, which contains a subtle flaw:

```
1 def popleft(lst):
2     element = lst[0]
3     lst = lst[1:]
4     return element
5
6 states = ['solid', 'liquid', 'gas', 'plasma']
7 popleft(states) → 'solid'
8 states → ['solid', 'liquid', 'gas', 'plasma']
```

Line three performs the same operation but it does so by constructing a new list rather than changing the existing list. Since we didn't change the list passed into `popleft()`, the `states` variable remains unchanged after calling `popleft()`<sup>3</sup>.

---

1. The C language uses call-by-value to pass arguments by copying each value into the parameter. Pointers can be used to approximate call-by-object.

2. Note that the `popleft()` method is provided in a more efficient implementation by Python's `deque` class.

3. In languages that support call-by-reference this code would have worked. Perl is one of the few

*Positional Arguments*

The most familiar syntax for passing arguments is to simply pass them into parameters in the same positional order they appear in the function definition.

The positional argument syntax is clearly demonstrated in this example:

```

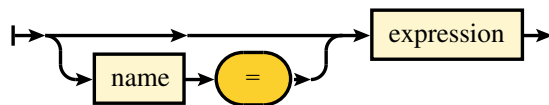
1 def evaluate_quadratic(x, a, b, c):
2     return a * x ** 2 + b * x + c
3
4 evaluate_quadratic(5, 3, 2, 1) → 86

```

The variable `x` is associated with 5, `a` with 3 and so on because the position of the call arguments matches with the order and number of parameters.

*Keyword Arguments*

When keyword arguments are used in a function call, the caller identifies the arguments by the parameter name.

**argument**

This language feature allows you to refer to arguments out of order because Python is able to use the parameter names to match the values with parameters. Lets rewrite the last example with keyword arguments:

```

1 def evaluate_quadratic(x, a, b, c):
2     return a * x ** 2 + b * x + c
3
4 evaluate_quadratic(c=1, b=2, a=3, x=5) → 86

```

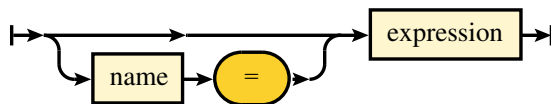
The interpretation of the last example is exactly the same as the positional argument example. Despite the fact that the arguments are given in reverse order, they are mapped to the same parameters. Creating good names for parameters is crucial in designing intuitive functions. In the last example `a`, `b`, and `c` were good choices because they are the common names for the coefficients of a polynomial.

---

widespread programming languages that supports call-by-reference as its default calling mechanism. Some languages, like C, support call-by-reference with special syntax.

*Default Arguments*

Arguments in Python can also have default values. When a parameter is written with a default argument, Python assumes the associated argument takes the default value if no value is explicitly provided in the function call.

**parameter**

Consider this variation on our `evaluate_quadratic` function:

```

1 def evaluate_quadratic(x, a=0, b=0, c=0):
2     return a * x ** 2 + b * x + c
3
4 evaluate_quadratic(5) → 0
5 evaluate_quadratic(5, 2) → 50
6 evaluate_quadratic(5, c=4) → 4

```

In the last example, the only parameter without a default value is `x`. Since it has no default, it is called a **required argument**.

*Variadic Functions*

Some of the functions we have used in this text are variadic, meaning they take a variable number of arguments. If we prefix a function's parameter with a `*` then the variable captures all of the unnamed arguments in a tuple (a data structure describe in more detail in the next chapter):

```

1 def average(*args):
2     return sum(args) / len(args)
3
4 average(1, 2, 3, 4) → 2.5

```

Python also lets us do the same thing for named parameters with a `**` prefix. Named parameters are captured in a `dict` (a data structure also described in more detail in the next chapter). Capturing both kinds of arguments at the same time gives us the general signature used in this example:

```

1 def arg_counter(*list_args, **keyword_args):

```

```

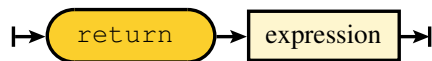
2     return (len(list_args), len(keyword_args))
3
4 arg_counter(1, 2, 3, a=4, b=5) → (3, 2)

```

### 5.3 Return Values

Every function has a return value which is passed to the caller. The return value is set by the `return` statement, which also exits the function at the same time.

#### return statement



Although functions may only return one value<sup>4</sup>, functions may have multiple `return` statements and this can be a powerful mechanism for controlling the flow of execution of a program. Consider this example where the `return` statement on line 3 creates an implicit `else` condition because line 4 will only be executed if line 2 evaluates to `False`:

```

1 def clamped_square(x):
2     if x < 0:
3         return 0
4     return x ** 2
5
6 clamped_square(-1) → 0
7 clamped_square(2) → 4

```

If a `return` statement omits a return value, the value `None` is returned implicitly.

```

1 def do_nothing():
2     return
3
4 do_nothing() → None

```

Likewise, if a function reaches the end of its evaluation without reaching a `return` statement, the function also returns `None`.

---

4. A single return value isn't really a limitation because that one value can be almost anything including a collection of other values.



```
1 def do_nothing():
2     pass
3
4 do_nothing() → None
```

`return` statements always return a single value, but keep in mind that value may be almost anything including `None` or collections of other values (e.g., lists, dictionaries, tuples).

#### 5.4 Docstrings

A **docstring** is a special string literal designed to document user defined abstractions in Python. Unlike a comment, a docstring becomes a part of the code and data that makes up your program. Specifically, it becomes an attribute called `__doc__` of the construct to which it is attached. Consider this example where we define and then print a docstring associated with a function:

```
1 def clamped_square(x):
2     """Return the square of x if x is positive;
3     return 0 otherwise."""
4     if x < 0:
5         return 0
6     return x ** 2
7
8 help(clamped_square)
```

Output:

```
Help on function clamped_square in module __main__:
```

```
clamped_square(x)
    """Return the square of x if x is positive;
    return 0 otherwise.
    (END)
```

Python aware development environments (e.g., IDLE, PyCharm) can use docstrings to provide programmer documentation as you work. And besides providing documentation to programmers who may use our function later, docstrings also help us describe our functions to ourselves.

**Principle 3** (Functions Should Do One Thing). *If you can't plainly and precisely describe the one thing that a function does, the function is probably poorly defined or overly complex.*

Docstrings can be used as a point of reflection and if your function requires complicated documentation that may be a sign that your program organization could be improved. The best docstrings are short and sweet. While these aren't hard rules many developers follow these common docstring conventions[41]:

- As a matter of consistency always use triple quotes for docstrings even if it is a one line string,
- The docstring summary should begin with an action verb (e.g., “Return this..”, “Do that..”)
- The docstring summary should end with a period.
- For multi-line documentation, the docstring should begin with a one line summary, then one blank line, and then the longer documentation.

## 5.5 Scope

Whenever a function is called this also creates a new **scope** - a context where identifiers can be defined. Scopes are nested such that when Python looks up an identifier name it always checks the current scope first, then checks any enclosing scopes (inner to outer)[42]. Python has three different kinds of scope:

1. the **built-in scope**, which is made up identifiers defined by Python,
2. the **global scope**, which is made up identifiers defined in a Python file but not within a function, and
3. the **function scope**, which is made up identifiers (including parameters) defined in a function.

Consider this simple example where these three different scopes are being used:

```
1 clamp_min = 0
2 clamp_max = 10
3
4 def clamp(value):
5     return min(max(value, clamp_min), clamp_max)
```

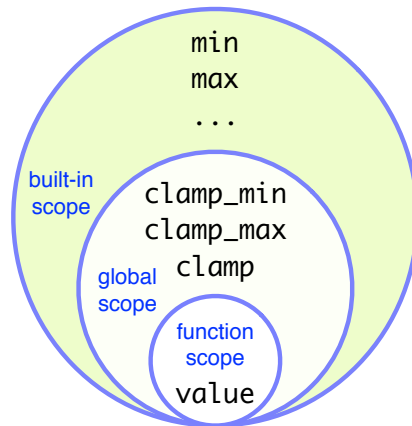


Figure 51: A visualization of the three principle scopes in Python - built-in identifiers form the outermost scope followed by global identifiers and function identifiers.

If we draw where each one of these variables in the last example falls in scope relative to line 5 the result would look something like Figure 51. When Python looks up the identifier `min` on line 5, it first checks if it is in the function scope. Since it isn't, it then checks the global scope. Since `min` isn't defined there either it finally checks the built-in scope. Since `min` is defined in the built-in scope, Python uses the definition for `min` that it finds in that scope. Python uses this same lookup process for all identifiers. Consider this example where the scope is vitally important:

```

1 seq_1 = ["A"]
2 seq_2 = ["A"]
3
4 def yen():
5     seq_1 = ["A", "G"]
6     seq_2.append("G")
7
8 yen()
9 seq_1 → ["A"]
10 seq_2 → ["A", "G"]

```

This code may not behave how you expect! After evaluating `yen()`, the `seq_1` variable contains only the value "A" (and not "A" followed by "G")! The reason is because there are two different identifiers in different scopes with

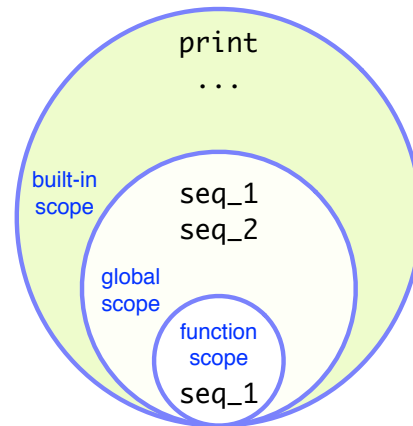


Figure 52: In this example the function scope has a variable `x` which shadows the global scope. The result is two different variables that happen to have the same name.

the same name, `seq_1`. For clarity, we have drawn the scopes relative to the `yen()` function in Figure 52.

When two variables in different scopes have the same name we call this **variable shadowing**. It's important to understand that as far as Python is concerned these are unrelated variables which may have different values and types. Scope also implies that variables have a lifetime where they fall in and out of scope as Python evaluates a program.

In Python, variables are defined and attached to a scope when they are first assigned. While this is generally pretty convenient, the consequences to scoping may, at first, seem inconsistent. Although `seq_1` only contains "A", `seq_2` does in fact contain both "A" and "G". That's because the assignment of `seq_1` inside `yen()` shadows the global `seq_1`. But since `seq_2` is not assigned within `yen`, Python recognizes it as the global `seq_2`. In sum, this piece of code has two `seq_1` variables and one `seq_2` variable!

But what if we really wanted only one `seq_1`? One mechanism Python has for crossing scopes is the `global` statement<sup>5</sup>. We can change the last piece of code to declare `x` as a `global`:

```
1 seq_1 = ["A"]
```

5. The same scope resolution problem can happen in nested functions as well. The `nonlocal` statement allows variables to cross scopes in much the same way `global` does [46].

```

2 seq_2 = ["A"]
3
4 def yen():
5     global seq_1
6     seq_1 = ["A", "G"]
7     seq_2.append("G")
8
9 yen()
10 seq_1 → ["A", "G"]
11 seq_2 → ["A", "G"]

```

And now this piece of code only has one `seq_1` variable and one `seq_2` variable! Well written Python code tends to avoid using global variables. Besides the fact that the Python syntax makes it difficult to distinguish local and global variables, functions that depend on global state often aren't as general and reusable as those that don't. In Chapter 9 we'll discuss how shared state can be represented using classes and objects, which is often preferable to using global variables.

## 5.6 Code Repetition

Lets write a program that recites the lyrics to the song "This Old Man." We might naively write it as such:

```

1 print("This old man, he played one,")
2 print("He played nick-knack on my thumb;")
3 print("With a nick-knack paddywhack,")
4 print("Give the dog a bone,")
5 print("This old man came rolling home.")
6 print("")
7
8 print("This old man, he played two,")
9 print("He played nick-knack on my shoe;")
10 print("With a nick-knack paddywhack,")
11 print("Give the dog a bone,")
12 print("This old man came rolling home.")
13 print("")
14
15 # and so on.. 8 more verses are recited

```

```

16 # where knick-knack is played on a tree,
17 # door, hive, sticks Devon, gate, line,
18 # and hen.

```

In total there would be 60 lines of code (including empty strings). The whole thing was so tedious that we only wrote the first two verses! It's clear there is a lot of repetition here. One task we can accomplish with functions is reducing code repetition.

**Strategy 2** (Identify and Factor Out Repetition with Functions). *Many problems have repetitive patterns that can be parameterized by a simpler pattern.*

Each verse of this song only differs by two words - the number and where knick-knack is played. If we think of a verse as a function then these two words would be the parameters to the function. With this in mind, we can rewrite the whole song:

```

1 def verse(number, place):
2     """Sing a verse of This Old Man with the given
3     number and place."""
4     print("This old man, he played ") + number + ",")
5     print("He played knick-knack on my ") + place + ";"
6     print("With a knick-knack paddywhack,")
7     print("Give the dog a bone,")
8     print("This old man came rolling home.")
9     print("")
10
11 verse("one", "thumb")
12 verse("two", "shoe")
13 verse("three", "tree")
14 verse("four", "door")
15 verse("five", "hive")
16 verse("six", "sticks")
17 verse("seven", "Devon")
18 verse("eight", "gate")
19 verse("nine", "line")
20 verse("ten", "hen")

```

Output :

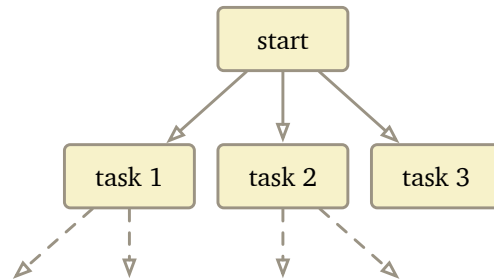


Figure 53: A top down design starts by breaking the problem up into large tasks and then those problems are broken up into subtasks and so on.

You know how this song goes!

While we have removed one kind of repetition from this example it can still be improved by factoring out the repetitive calls to `verse`. We will consider this in Chapter 7 Iteration.

## 5.7 Top-Down Design

Many of the problems we have as computer scientists are complex. One way to manage this complexity is to break the problem into smaller pieces. We can start at the most abstract level (the top) and begin implementing functions until it doesn't make sense to break up the problem further (the bottom). This process of breaking problems up until we have manageable sub-problems is called **top-down design**.

Functions provide a convenient organizing unit in top-down design because we can associate problems and sub-problems with functions which represent the associated solutions. As we begin doing top-down design we don't immediately worry about functions that work but instead concentrate on the relationship between functions as they represent solutions to problems and associated sub-problems.

Lets use Python to draw a landscape. Our landscape will have lots of features but rather than thinking about which line to draw first, lets break the landscape up into sub-drawings:

```
1 REPLACE WITH GRAPHIC
2 def draw_landscape():
3     draw_mountains()
```

```
4     draw_sun()
5     draw_trees()
```

Then we break up each one of these functions into sub-functions as appropriate. When we have a function that we can't break down any more (e.g., `draw_sun()`) then we solve that one subproblem. Here's our solution to this problem:

```
1  import turtle
2
3  pen = turtle.Turtle()
4  pen.up()
5  pen.pensize(3)
6
7  def draw_landscape():
8      draw_mountains()
9      draw_sun()
10     draw_trees()
11
12  def draw_mountains():
13     draw_mountain()
14     pen.forward(200)
15     draw_mountain()
16     pen.backward(200)
17
18  def draw_trees():
19     draw_tree()
20     pen.forward(100)
21     draw_tree()
22     pen.forward(100)
23     draw_tree()
24     pen.backward(200)
25
26  def draw_tree():
27     pen.down()
28     pen.left(90)
29     pen.forward(50)
30     pen.right(90)
31     pen.forward(50)
```



```
32     pen.left(45)
33     # ...
34
35 def draw_mountain():
36     # ...
37
38 def draw_sun():
39     # ...
```

By breaking this big project into function generated with a top-down approach, we have given the code a governing organization and readability. This strategy has a number of advantages over considering the problem as a whole:

1. If we break the problem up into smaller parts, we clarify the tasks that need to be done to solve the problem.
2. Each smaller part is typically less complicated than the whole, making the problem more approachable.
3. Smaller parts may be reusable, reducing the implementation size.
4. Breaking the problem up also allows us to distribute the problem solving work, enabling team work.

## 5.8 Recursion \*

When a function calls itself, we call the resulting execution flow **recursion**. It's very easy to make a function call itself but making useful recursive functions requires some extra care.

Consider the problem of summing the natural numbers from 1 to  $n$ . A recursive solution to this problem might look like this:

```
1 def sum_to(n):
2     """Return the sum of the natural numbers from 1 to n."""
3     if n == 1:
4         return 1
5     return sum_to(n-1) + n
6
7 sum_to(100) → 5050
```

A common structure for recursive functions includes:

- base case(s) that return a non-recursive result,
- recursive case(s) that break the problem into a smaller problem.

If a recursive function of this kind doesn't have sufficient base cases or the recursive cases don't cause the function to move toward a base case the function can recurse endlessly causing **infinite recursion**. Another concern in using recursion is the total cost of calling functions recursively (each function call uses a small but significant amount of memory and time).

Many mathematical formulas are inherently recursive. Consider this common definition for a factorial:

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ (n-1)! \times n & \text{otherwise} \end{cases}$$

Notice that the mathematical function begins with a base case and then defines a recursive case. The same function implemented in Python becomes:

```

1 def factorial(n):
2     """Return the factorial of n."""
3     if n == 1:
4         return 1
5     return factorial(n-1) * n
6 factorial(10) → 3628800

```

Recursion has a number of natural applications in Computer Science - it's often used to traverse graphs and trees, parse languages, and process lists. In fact, recursion can be used for any iterative or repetitive task. In Chapter 7 we will discuss other control structures for implementing iteration in Python.

## 5.9 Functions as Values \*

When a function can be used just like any other value in the language - we call it a **first-class function**<sup>6</sup>. Python supports first-class functions, meaning that functions can behave like values and function names are like any other variable name. This is easy to illustrate:

```

1 def hello():

```

---

6. The term first-class function was coined by Christopher Strachey in the 1960s to describe functions which could be treated as values [37]. First-class functions which use lexical scoping rules was an innovation introduced in the Scheme language in 1975 [45].

```

2     return "Hello"
3
4     greeting = hello
5     greeting() → "Hello"

```

In this bit of code we defined a function `hello`, then we defined a variable `greeting` and set its value to that of `hello` (a function). Then calling `greeting()` evaluates the function we originally associated with `hello`. As you can see parenthesis are extremely important in differentiating a function call from the function itself.

We can do more than simple variable name trickery, we can also pass functions around like any other value:

```

1 import math
2
3 def make_dx(f, delta_x=0.000001):
4     """Return a function that approximates the
5         derivative of f."""
6     def deriv(x):
7         return (f(x + delta_x) - f(x)) / delta_x
8     return deriv
9
10 sin_dx = make_dx(math.sin)
11 sin_dx(1.2) → 0.36235728850808613
12 math.cos(1.2) → 0.3623577544766736

```

Here we have passed the function `math.sin` into the `make_dx` function and `make_dx` returned the function `deriv`!

A careful reader might also note that the function `deriv` uses the parameter `delta_x` well after it should have fallen out of scope! When a function uses variables from a parent scope it forms a **closure**<sup>7</sup>. The variables in the closure are attached to the function and will continue to live as long as the function does.

## Glossary

**arguments** The data passed to a function when it is called.

---

7. The term closure was coined by British computer scientist Peter Landin who used it describe how a function's free variables (called open bindings in lambda calculus) are bound to the lexical environment (thus, closing the bindings) [43].

- built-in\_scope** Identifiers defined by the Python language.
- call-by-reference** A technique for passing parameter values to a function where a reference to a value is copied instead of the value itself.
- call-by-value** A technique for passing parameter values to a function where the actual values are copied in whole and then assigned to the parameters.
- closure** A function combined with the variables the function references.
- default\_arguments** If a parameter specifies a default value, the argument is optional and does not have to be passed a value.
- docstring** A special string at the beginning of many Python entities that describes the entity and can be accessed programmatically.
- first-class\_function** A linguistic feature where functions can be treated like values (e.g., passed to other functions, returned from functions and stored in variables).
- function\_scope** Identifiers defined within a function.
- global\_scope** Identifiers defined by the programmer at the top most level of a file.
- keyword\_arguments** A mapping of arguments to parameters based on the names of the parameters.
- parameter** A special variable that names the values called with a function.
- positional\_arguments** A mapping of arguments to parameters based on the position of the arguments.
- return\_value** Every function returns a value that represents the result of the function's computation.
- scope** A context where identifiers are defined.
- top-down-design** The process of breaking up problems from a high-level to a low-level such that we define manageable subproblems.
- variable\_shadowing** Occurs when two variables have the same name but different scopes.

**Reading Questions**

- 5.1. What are the significant structural elements that define a function?
- 5.2. What happens when a function has finished evaluating?
- 5.3. What happens when a return statement is reached?
- 5.4. How are parameters related to arguments?
- 5.5. Which parameter passing technique does Python use?
- 5.6. Why are keyword arguments useful?
- 5.7. Why are default arguments useful?
- 5.8. If a function doesn't explicitly return a value, what is returned?
- 5.9. How are docstrings different from comments?
- 5.10. What are some conventions Python programmers observe when composing docstrings?
- 5.11. What are the three kinds of scope in Python?
- 5.12. What is variable shadowing?
- 5.13. What does the global statement do?
- 5.14. Explain Python's scoping rules.
- 5.15. What kind of problem would you apply top-down-design to?
- 5.16. What are some advantages to top-down design?
- 5.17. What is recursion?
- 5.18. What two cases are common in recursive function design?
- 5.19. What is a first-class function?
- 5.20. What's wrong with code repetition in a program?

**Exercises****5-1. Skip to My Lou**

Write a program that prints the lyrics to "Skip to My Lou." Use function(s) to reduce the repetition in the song. The lyrics to the song are:

```

1 Skip, skip, skip to my Lou,
2 Skip, skip, skip to my Lou,
3 Skip, skip, skip to my Lou,
4 Skip to my Lou, my darling.
5 (Changing verse)
6 (Changing verse)
7 (Changing verse)
8 Skip to my Lou, my darling.

```

This is repeated a total of seven times with these changing verses:

- a) Fly's in the buttermilk, Shoo, fly, shoo
- b) There's a little red wagon, Paint it blue
- c) Lost my partner, What'll I do?
- d) I'll get another one, Prettier than you
- e) Can't get a red bird, Jay bird'll do
- f) Cat's in the cream jar, Ooh, ooh, ooh
- g) Off to Texas, Two by two

### 5-2. Perimeter of a triangle

Define a function that given the sides of a triangle computes and displays its perimeter. The perimeter of a triangle is given by  $P = a + b + c$ .

Test code:

```

1 print(perimeter(3, 3, 3))
2 print(perimeter(10, 11, 12))
3 print(perimeter(100.1, 200.1, 300.1))

```

Sample run:

```

1 9
2 33
3 600.3

```

### 5-3. Even or odd

Define a function that when passed a number returns whether that number is even or odd.

Test code:

```

1 print(is_odd(3))
2 print(is_odd(4))
3 print(is_odd(-1))

```

Sample run:

```
1 True
2 False
3 True
```

#### 5-4. Prime numbers

Define a function that when passed a number determines if that number is prime. Remember a prime number is a natural number greater than one that has no positive divisors other than one and itself.

Test code:

```
1 print(is_prime(3))
2 print(is_prime(4))
3 print(is_prime(7))
4 print(is_prime(23))
5 print(is_prime(42))
```

Sample run:

```
1 True
2 False
3 True
4 True
5 False
```

#### 5-5. Same last digit

Define a function that takes in three numbers and returns `True` if they all have the same last digit.

Test code:

```
1 print(same_last_digit(2, 142, 262))
2 print(same_last_digit(333, 36, 43))
3 print(same_last_digit(41, 1, 21))
```

Sample run:

```
1 True
2 False
3 True
```

#### 5-6. Second highest

Define a function that given three numbers returns the second highest number.

Test code:

```
1 print(second_highest(900.5, 1000, 65))
2 print(second_highest(333, 36, 43))
3 print(second_highest(12, 2, 8))
```

Sample run:

```
1 900.5
2 43
3 8
```

### 5-7. Zodiac sign

Define a function that given a month and a day returns the corresponding Zodiac sign. The zodiac signs and dates can be found at <http://en.wikipedia.org/wiki/Zodiac>.

Test code:

```
1 print(zodiac_sign(9, 22))
2 print(zodiac_sign(2, 30))
3 print(zodiac_sign(7, 22))
```

Sample run:

```
1 Virgo
2 Aquarius
3 Cancer
```

### 5-8. Palindromes

A palindrome is a word or sentence that when read backwards says the same thing as when read forwards (e.g., “racecar”). Write a function that checks if a string is a palindrome without using iteration.

Test code:

```
1 print(is_palindrome("racecar"))
2 print(is_palindrome("carrace"))
3 print(is_palindrome("neveroddoeven"))
```

Sample run:

```
1 True
2 False
3 True
```

### 5-9. Intersection of two lines

Write a program that takes the slope and y intercept of two lines  $y = mx + b$ , finds and displays their point of intersection, or if the lines are parallel, or do not intersect. Recall the formula for a straight line is  $y = mx + b$ , where  $m$  is the slope and  $b$  is the y-intercept.

Test code:

```
1 print(line_intersection(-.5, 7, 2, 3))
2 print(line_intersection(3, 4, -2, 3))
3 print(line_intersection(2, 21, 2, 5))
```



Sample run:

```
1 (1.6, 6.2)
2 (-0.2, 3.4)
3 lines are parallel
```

### 5-10. Time since midnight II

As in exercise 3-10, write a program to calculate the time since midnight in seconds. This time, define a function that takes in a time as a string, i.e.: '04:30:46' and return the seconds from midnight.

Test code:

```
1 print(time_since_midnight('4:30:46'))
2 print(time_since_midnight('1:26:15'))
3 print(time_since_midnight('8:30:01'))
```

Sample run:

```
1 16246
2 5175
3 30601
```

### 5-11. Point in circle

An equation for a circle is  $(x - C_x)^2 + (y - C_y)^2 = r^2$ . Write a program that can tell if a point is within the boundaries of a circle. Your program should use a function that takes in the point you wish to test  $(x, y)$ , the center point of a circle  $(C_x, C_y)$ , and its radius. Return True or False as the result.

Test code:

```
1 print(point_in_circle((2, 3), (1, 2), 3))
2 print(point_in_circle((-2, 0), (1, 2), 3))
3 print(point_in_circle((0, 4), (1, 2), 3))
```

Sample run:

```
1 True
2 False
3 True
```

### 5-12. Point in rectangle

Write a program that given a rectangle ABCD and a point (x,y) can determine if the point lies within the rectangle. Use functions in your program to check your point. Display True or False as a result.

Test code:

```

1 rectangle = [(-2, 1), (0,3), (3, 0), (1, -2)]
2 point_in_rectangle(rectangle, (0, -2))
3 point_in_rectangle(rectangle, (1, 0))
4 point_in_rectangle(rectangle, (0, 2))

```

Sample run:

```

1 False
2 True
3 True

```

### 5-13. Linear equation solver (using Cramer's rule)

Write a program that solves systems of linear equations using Cramer's rule. Recall Cramer's rule is a method for solving a linear system of equations using determinants where  $x = \frac{D_x}{D}$ ,  $y = \frac{D_y}{D}$ ,  $z = \frac{D_z}{D}$ . Your program only needs to work with two equations in the form  $ax+by = c$ . Define a function that takes the parameters (a1, b1, c1, a2, b2, c2).

Test code:

```

1 print(linear_eq_solver(3, -4, 1, 5, 1, 17))
2 print(linear_eq_solver(-2, 6, 12, 1, -7, 14))
3 print(linear_eq_solver(4, 5, 2, -1, 5, 3))

```

Sample run:

```

1 (3.0, 2.0)
2 (-21.0, -5.0)
3 (-0.2, 0.56)

```

### 5-14. Compound interest calculator II

As in exercise 4-15, write a program to calculate compound interest. This time define a function that takes in the parameters (p, r, n, t) and returns the result, principal + interest. The formula for compound interest is:  $A = P(1 + \frac{r}{n})^{nt}$

Test code:

```

1 print(compound_interest_calc(1000, .04, 1, 2))
2 print(compound_interest_calc(5000, .06, 1, 4))
3 print(compound_interest_calc(1500, .10, 1, 1))

```

Sample run:

```

1 1081.60
2 6312.38
3 1650.00

```

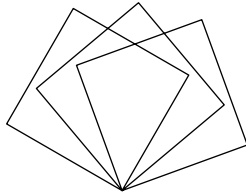
**5-15. Tilted squares**

Using Python's turtle graphics replicate the picture below. Define a function that draws the square shape given the edge length, and tilts each square drawn by a given angle.

Test code:

```
1 tilted_squares(100, 20)
```

Sample run:

**5-16. ISBN-10 validator**

Create a function that takes in an ISBN-10 number and checks the checksum digit to validate the number. Refer to exercise 4-16, where we computed the checksum digit. If the ISBN-10 number is valid return True, otherwise return False.

Test code:

```
1 print(isbn_checker('1430232374'))
2 print(isbn_checker('0262312193'))
3 print(isbn_checker('0321680561'))
```

Sample run:

```
1 True
2 False
3 True
```

**5-17. Writing numbers**

In exercise 4-11 we wrote numbers out as english. This time define a function that takes in a two digit number and returns the number written in English. For simplicity your program only needs to handle the number range 0-99.

Test code:

```
1 print(convert_to_english(5))
2 print(convert_to_english(20))
3 print(convert_to_english(35))
```

Sample run:

```

1 Five
2 Twenty
3 Thirty-Five

```

### 5-18. Temperature conversion

In question 3-7 we covered converting Fahrenheit to Celsius. This time implement a temperature conversion function that takes in a temperature, whether it is in Fahrenheit, Celsius, or Kelvin, and what to convert it to. Return your results formatted to 1 decimal place.

Test code:

```

1 print(temp_conversion(90, 'F', 'C'))
2 print(temp_conversion(-4, 'C', 'F'))
3 print(temp_conversion(10, 'C', 'K'))

```

Sample run:

```

1 32.2
2 24.8
3 283.0

```

### 5-19. Cubic function

Design a program to solve cubic equations. In your program implement a function `cubic_solver(a, b, c, d)` that solves the equation, and returns the roots. Remember a cubic equation has the form  $ax^3 + bx^2 + cx + d = 0$ .

Test code:

```

1 print(cubic_solver(1, -6, 11, -6))
2 print(cubic_solver(-4, 53, -130, 0))
3 print(cubic_solver(2, 2, 2, 10))

```

Sample run:

```

1 (1, 2, 3)
2 (10, 8.88e-16, 3.25)
3 (-1.88, 0.44 + i* 1.56, 0.44 - i* 1.57)

```

### 5-20. Recursive tree

Using Python's turtle package create a function that recursively draws a tree. Think about how you are going to draw the tree recursively. Do you draw it all at once, left side first, or right side first? What is the base case going to be that stops the recursion? The function `draw_tree()` will call your function `tree(branch_len, t)`, taking in the length you want for the tree's branches and the name you assign turtle. The result of your drawing should look like the figure below.

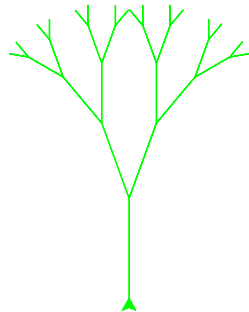
You can use the following code to get started:

```
1 import turtle
2
3 def tree(branch_len,t):
4     # Your code here.
5
6 def draw_tree():
7     t = turtle.Turtle()
8     my_tree = turtle.Screen()
9     t.left(90)
10    t.up()
11    t.backward(100)
12    t.down()
13    t.color("green")
14    tree(80,t)
15    my_tree.exitonclick()
16
17 draw_tree()
```

Test code:

```
1 draw_tree()
```

Sample run:



## Chapter Notes

In the early 1930s Alonzo Church developed lambda calculus as a formal system for describing computation using functions and variable bindings. John McCarthy built on these mathematical formalism in the design of LISP, which was first described in 1958. User defined functions were added to the FORTRAN language the same year [44].

Before functions were available in high-level programming languages, programmers would jump to specific points within the code - often using a statement called 'goto'. Unlike a function, a goto is one-way and lacks arguments or any scope boundaries.

In 1966 Bohm and Jacopini proved that sequences of function calls, selection structures (e.g., if statements), and iteration structures (e.g., while or for statements) could describe any computable function. This result was followed by Edsger Dijkstra's 1968 letter "Go To Statement Considered Harmful" which advocated for a **structured programming** paradigm that emphasized control structures with clear and easily traceable consequences.

Two paradigms directly descended from structured programming are procedural programming and functional programming. In **procedural programming**, functions are treated as procedures that take actions and may change the state of the program. In **functional programming**, functions act like mathematical functions (often modeled on Church's lambda calculus) that return values but don't change the state of a program directly.

Because changes in state are limited in functional programming it's often easier to reason about functional code. For readers who want to explore functional programming in more depth we recommend Friedman and Felleisen's "The Little Schemer," which is both humorous and enlightening [40].

First-class functions with lexical scoping were first defined and popularized in the Scheme language in 1975 [45] and have become an extremely common feature in programming language design. You can find first-class functions in Common Lisp, Haskell, Scala, JavaScript, PHP, Perl, Python, Ruby, Mathematica and many others. Limited support for first class functions has even appeared in the C++11 standard, the Objective-C blocks standard, and the Java 8 standard.

One functional form we won't discuss in any depth in this text is Python's `lambda` operation, which creates an unnamed function within an expression. Lambda functions are common in functional programming languages but their syntax in Python is crippled. A lambda function in Python is limited to a single expression. In this example we create a function that squares a value:

```
1 square = lambda x: x ** 2
2 square(3) → 9
```

We gave the function the name `square()` without a function declaration in the last example, but the real point is not to name the function at all. In this example, we create the function and apply it to multiple values in a single

step using `map()`:

```
1 map(lambda x: x ** 2, [1, 2, 3, 4])
2     → [1, 4, 9, 16]
```

Lambda functions are problem solving workhorses in languages like Common Lisp, Scheme, and JavaScript but they lack the same power and utility in Python.

- [36] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. **Communications of the ACM**, 9(5):366–371, May 1966. doi: [10.1145/355592.365646](https://doi.org/10.1145/355592.365646). [link](#).
- [37] Rod Burstall. Christopher Strachey — understanding programming languages. **Higher-Order and Symbolic Computation**, 13(1-2):51–55, 2000. doi: [10.1023/A:1010052305354](https://doi.org/10.1023/A:1010052305354).
- [38] Felice Cardone and J Roger Hindley. **Handbook of the History of Logic, Volume 5**, chapter History of Lambda-calculus and Combinatory Logic. North Holland, 2009. isbn: [0444516204](https://www.isbn-international.org/product/0444516204).
- [39] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. **Communications of the ACM**, 11(3):147–148, March 1968. doi: [10.1145/362929.362947](https://doi.org/10.1145/362929.362947).
- [40] D.P. Friedman and M. Felleisen. **The Little Schemer**. MIT Press, 1996. isbn: [9780262560993](https://www.isbn-international.org/product/9780262560993).
- [41] David Goodger and Guido van Rossum. PEP257 - Docstring conventions. **Python Enhancement Proposal**. [link](#).
- [42] Jeremy Hylton. PEP227 - Statically nested scopes. **Python Enhancement Proposal**. [link](#).
- [43] Joel Moses. AIM199 - The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. Technical report, Cambridge, MA, 1970. [link](#).
- [44] Robert W. Sebesta. **Concepts of Programming Languages**. Addison-Wesley Publishing Company, 9th edition, 2009. isbn: [0136073476](https://www.isbn-international.org/product/0136073476).
- [45] Guy L Steele and Gerald J Sussman. Lambda: The ultimate imperative. Technical report, Cambridge, MA, USA, 1976. [link](#).

- [46] Ka-Ping Yee. PEP3104 - Access to names in outer scopes. **Python Enhancement Proposal**. [link](#).



## 6

# Collections

*As it seems to me, in Perl you have to be an expert to correctly make a nested data structure like, say, a list of hashes of instances. In Python, you have to be an idiot not to be able to do it, because you just write it down.*

*Peter Norvig*

```
1 overview = ["Lists",
2             "Dictionaries",
3             "Tuples",
4             "Sets",
5             "Glossary",
6             "Reading Questions",
7             "Exercises",
8             "Chapter Notes"]
```

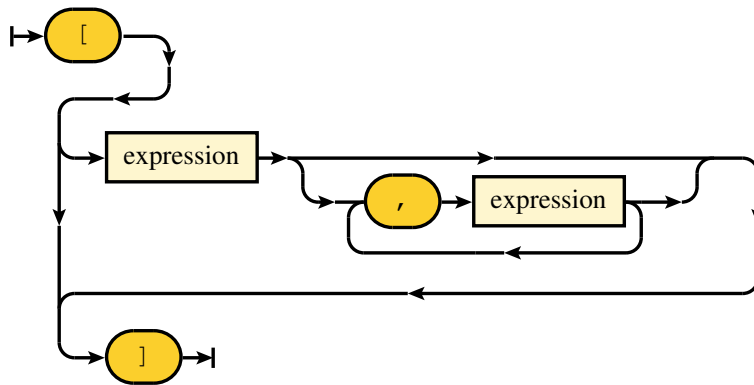
### 6.1 Lists

A Python **list** isn't so different from a list you might make to go to the grocery store. A list is simply an ordered sequence of objects. Lists are conceptually similar to mathematical vectors, matrices, and tensors which store ordered numeric values. Lists are also used like arrays, a data structure common in languages like C, C++, and Java which can also store an ordered sequence of elements.

Lists can be created from constructors and method calls like other objects, but

they may also be formed using Python's list syntax by using two matched braces with comma separated values. Note that spaces, newlines and indentation between objects is insignificant inside a list construction.

### list literal



The values stored in a list are not constrained to any one type or any specific type. We can create a list of strings:

```
1 gang = ['Fred', 'Daphne', 'Velma', 'Shaggy', 'Scooby']
```

Or we can just as easily create a list of numbers:

```
2 exam_grades = [95.0, 92.0, 100.0, 85.0]
```

Due to Python's dynamic nature, lists may even contain values of unrelated types or have unrelated purpose.

```
3 unrelated = [42.0, True, 11, 'Mystery_Machine']
```

Storing mixed types in a list is sometimes useful (e.g., reading or writing rows from a spreadsheet or database) but it may also hint at a poor design. When you need to represent relationships between data you should consider using dictionaries or objects instead. As a rule of thumb, if you can't describe a list as a collection of one kind of thing your program design likely has a problem. As an example, consider these lists and what they store:

```
4 # GOOD: A list of US states
5 states = ['California', 'Oregon', 'Washington']
```

```

6
7 # GOOD: A list of years
8 years = [1850, 1859, 1889]
9
10 # POOR: A list of US states and years
11 states_and_years = ['California', 1850,
12                    'Oregon', 1859,
13                    'Washington', 1889]

```

The most common list methods and operations are summarized in Table 61.

Table 61: *Common List Operations*

Operation	Description
<code>x = [y_1, ..., y_n]</code>	Create a new list with items <code>y_1 ... y_n</code>
<code>x[i]</code>	Retrieve the value in <code>x</code> associated with index <code>i</code>
<code>x[i] = y</code>	Set the value in <code>x</code> associated with index <code>i</code> to <code>y</code>
<code>x[i:j]</code>	Create a new list with values associated with indexes <code>i</code> to <code>j</code> (exclusive)
<code>x.append(y)</code>	Append <code>y</code> to the list <code>x</code>
<code>x.insert(i, y)</code>	Insert <code>y</code> at index <code>i</code> within <code>x</code>
<code>del x[i]</code>	Remove the entry in <code>x</code> associated with index <code>i</code>
<code>len(x)</code>	Return the number of elements in <code>x</code>
<code>list(x)</code>	Produces a list from sequence <code>x</code>
<code>y in x</code>	Returns <code>True</code> if <code>y</code> is in the list <code>x</code> ; returns <code>False</code> otherwise
<code>x.sort()</code>	Sort the items in <code>x</code> in place
<code>x.reverse()</code>	Reverse the items in <code>x</code> in place
<code>x + y</code>	Concatenate the lists <code>x</code> and <code>y</code> to form a new list

Like strings, list elements are numbered sequentially starting at zero or using negative indexes from the end of the list.

0	1	2	3	4
'Fred'	'Daphne'	'Velma'	'Shaggy'	'Scooby'
-5	-4	-3	-2	-1

Using this convention we can access elements of a list much like we do for individual characters in a string.

```

1 gang = ['Fred', 'Daphne',
2         'Velma', 'Shaggy',
3         'Scooby']
4 gang[0] → 'Fred'
5 gang[-1] → 'Scooby'

```

Slicing lists also works much like it does with strings. Each slice operation produces a new sublist based on the starting index (inclusive) and ending index (exclusive) specified.

```

6 gang[0:2] → ['Fred', 'Daphne']
7 gang[2:] → ['Velma', 'Shaggy', 'Scooby']

```

But unlike strings, we can change the contents of a list through a variety of different methods or operators.

```

8 gang[-1] = 'Scooby_Doo'
9 gang.append('Scrappy_Doo')
10 gang → ['Fred', 'Daphne',
11         'Velma', 'Shaggy',
12         'Scooby_Doo', 'Scrappy_Doo']

```

Lists are **mutable**, meaning their values can change. This differs from most other data-types we have worked with up to this point. Strings, integers, floats, and Booleans are all examples of **immutable** types. We can change which immutable value a variable holds but we can't change the value itself. Consider this simple statement and its values:

```

1 x = 5
2 x += 1
3 x → 6

```

While we change the variable `x` in the second line, the values in this example are not changing. The integer value 5 is 5 and will always be 5. When we add 1 to 5, we aren't changing 5, instead Python creates a new integer value,

6 as a result of the addition operation. The same thing is true of Booleans, floats and even strings – we can create new strings but we never truly change existing strings. Lists are different - we can add, remove and change what elements make up a list.

### *Splitting Strings*

One really useful way of generating a list is by splitting a string into substrings. This is accomplished using the string split method.

```
1 "Nice work Scoob!".split() →  
2 ['Nice', 'work', 'Scoob!']
```

By default split uses whitespace to divide the substrings but different separators can be passed as an argument. This is a powerful way to pull strings a part into useful components.

```
1 "scooby@doo.com".split('@') →  
2 ['scooby', 'doo.com']  
3  
4 "11.2,22.7".split(',') →  
5 ['11.2', '22.7']
```

### *Lists of Lists*

Since lists contain objects, and lists are objects that means that we can create lists of lists. A list within a list is an example of a nested data structure. All the same operations and methods apply but the ramifications are powerful.

Lists can be constructed one inside another and their membership types and counts do not have to match. In this example we have a list of families.

```
1 families = [  
2   ['Gomez', 'Morticia', 'Pugsley', 'Wednesday'],  
3   ['Herman', 'Lily', 'Grandpa', 'Eddie'],  
4   ['Fred', 'Casey', 'Tina', 'Ike', 'Turner']]
```

If we wanted to print Herman, we would find him in the second outer list (index 1) and at the first position (index 0) in the inner list.

```
4 families[1][0] → 'Herman'
```

You can think of each pair of access operations (the square braces) as being evaluated similar to a function call. The first call returns the list at index 1.

The second call then returned the string value at index 0. Once you get used to the idea that a list may contain other lists manipulating them is pretty straight-forward.

```

5 # Lets add another family member.
6 families[0].append("Uncle_Fester")
7 families[0] →
8   ['Gomez', 'Morticia',
9    'Pugsley', 'Wednesday',
10   'Uncle_Fester']
11
12 # Or even add and delete whole families.
13 families.append(['Leland', 'Sarah', 'Laura'])
14 del families[0]
15 families →
16   [['Herman', 'Lily', 'Grandpa', 'Eddie'],
17    ['Fred', 'Casey', 'Tina', 'Ike', 'Turner'],
18    ['Leland', 'Sarah', 'Laura']]

```

Nested data structures are sometimes called **multi-dimensional** especially when each nested level can be thought of as a dimension for that data. A matrix, for example, is a two-dimensional data structure because the rows and columns each represent a dimension of the data. We could use lists to represent an identity matrix by nesting lists of individual rows:

```

1 identity = [[1.0, 0.0, 0.0],
2             [0.0, 1.0, 0.0],
3             [0.0, 0.0, 1.0]]

```

Accessing individual elements of the matrix uses standard list notation but beware that the outer list represents the row and the inner list represents the column.

```

4 identity[2][0] = 7.0
5 identity →
6   [[1.0, 0.0, 0.0],
7    [0.0, 1.0, 0.0],
8    [7.0, 0.0, 1.0]]

```

If we changed the numbers in the matrix to X's and O's we could represent a Tic-Tac-Toe board.

```

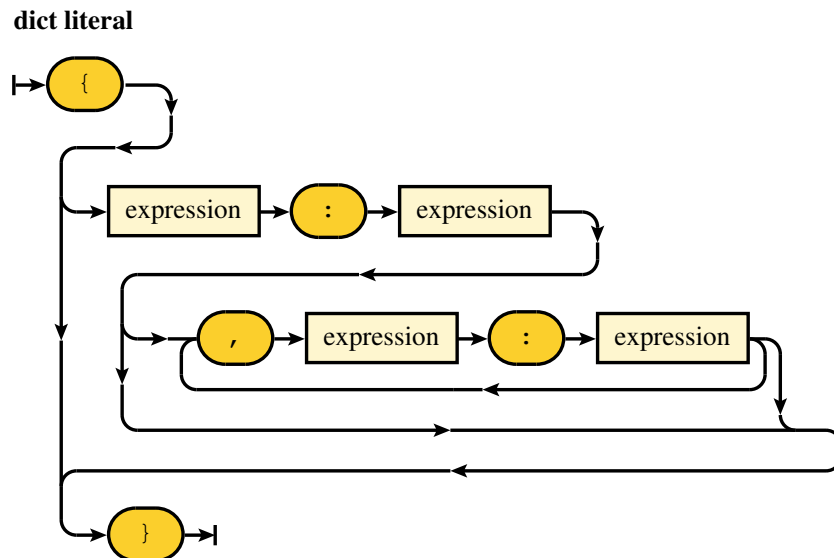
1 board = [['X', 'O', 'O'],
2         ['_', 'X', 'O'],
3         ['_', '_', 'X']]

```

Nested lists are a powerful mechanism for representing almost anything with data that can be organized in rows, columns, axes, or dimensions.

## 6.2 Dictionaries

**Dictionaries** are collections of named values. A dictionary can be constructed from paired curly braces that contain a list of key value pairs.



Dictionary keys (the first part of a key-value pair) can be any of the core types we discussed in earlier chapters including strings, numbers and Booleans but not mutable types like lists and dictionaries. Dictionary values (the second part of a key-value pair) can be of any type.

```

1 # Each member of the gang has a favorite color.
2 gang = {'Fred': 'blue',
3        'Daphne': 'violet',
4        'Velma': 'orange',

```

```

5         'Shaggy': 'green',
6         'Scooby': 'brown'}

```

We access the values of a dictionary by using the associated keys.

```

1 # How can we find out Scooby's favorite color?
2 gang['Scooby'] → 'brown'
3
4 # We can also add members dynamically.
5 gang['Scrappy'] = 'white'
6
7 # And we can change their favorite colors.
8 gang['Scooby'] = 'blue'

```

Dictionaries provide a powerful tool for organizing data and like lists, we can even nest dictionaries within each other!

```

1 gang = {'Fred': {
2         'color': 'blue',
3         'likes': 'traps'},
4        'Daphne': {
5         'color': 'violet',
6         'likes': 'mysteries'},
7        'Velma': {
8         'color': 'orange',
9         'likes': 'glasses'},
10       'Shaggy': {
11        'color': 'green',
12        'likes': 'sandwiches'},
13       'Scooby': {
14        'color': 'brown',
15        'likes': 'Scooby_□Snacks'}}
16
17 # What does Fred like?
18 gang['Fred']['likes'] → 'traps'
19
20 # What is Daphne's favorite color?
21 gang['Daphne']['color'] → 'violet'

```



Dictionaries provide an efficient means to lookup named information. Dictionaries have many methods and operations but some of the most common are summarized in Table 62.

Table 62: *Common Dictionary Operations*

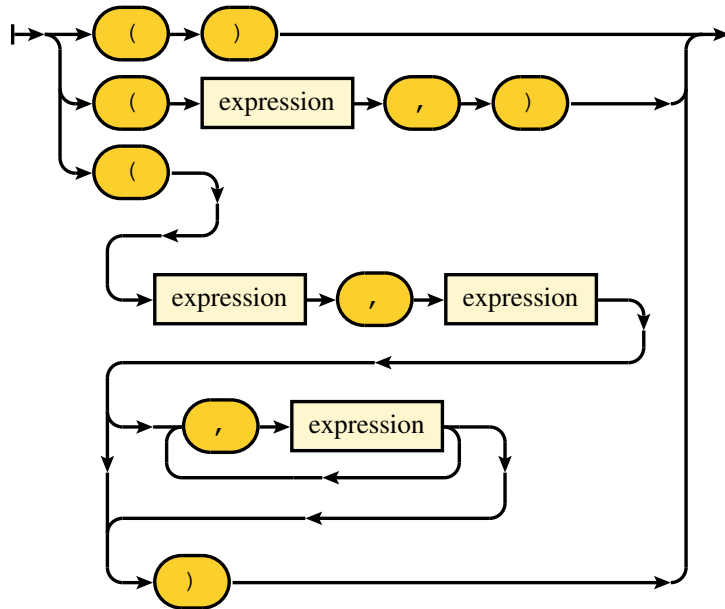
Operation	Description
<code>x = {k_1: y_1, ..., k_n: y_n}</code>	Create a new dictionary with key value pairs <code>(k_1, y_1) ... (k_n, y_n)</code>
<code>x[k]</code>	Retrieve the value in <code>x</code> associated with key <code>k</code>
<code>x[k] = y</code>	Set the value in <code>x</code> associated with key <code>k</code> to <code>y</code>
<code>del x[k]</code>	Remove the entry in <code>x</code> associated with key <code>k</code>
<code>len(x)</code>	Return the number of elements in <code>x</code>
<code>y in x</code>	Returns <code>True</code> if the key <code>y</code> is in the dictionary <code>x</code> ; Returns <code>False</code> otherwise
<code>x.items()</code>	Return an iterable sequence of key value pairs
<code>x.keys()</code>	Return an iterable sequence of keys
<code>x.values()</code>	Return an iterable sequence of values

### 6.3 Tuples

A tuple is an immutable ordered sequence of values. Tuples are essentially lists that you can't change. The fact that you can't change a tuple makes tuples inherently simpler than lists. While this might seem extremely constraining, tuples typically take up less memory and working with them may be faster than working with a list.

Tuples are formed like lists but using parenthesis instead of brackets:

## tuple literal



```

1 gang = ('Fred', 'Daphne', 'Velma', 'Shaggy', 'Scooby')
2 coord = (43.2, 28.5)
3 nested = ((1, 2), (3, 4))
4 empty_tuple = (

```

Using this syntax a tuple with only one member would be ambiguous because it would be in-differentiable from an expression surrounded by parenthesis. Thus, an oddball part of the Python syntax is that a single element tuple must have a trailing comma:

```

1 dogs = ('Scooby',)

```

Table 63: Common Tuple Operations

Operation	Description
<code>x = (y_1, ..., y_n)</code>	Create a new tuple with <code>y_1 ... y_n</code> .
<code>x[i]</code>	Retrieve the value in <code>x</code> associated with index <code>i</code> .
<code>x[i:j]</code>	Create a new tuple with values associated with indexes <code>i</code> to <code>j</code> (exclusive)

Operation	Description
<code>len(x)</code>	Return the number of elements in x.
<code>tuple(x)</code>	Produces a tuple from a sequence x.
<code>y in x</code>	Returns True if y is in the tuple x. Returns False otherwise.
<code>x + y</code>	Concatenate the lists x and y to form a new list.

Python has some syntactic shorthand for tuples such that commas outside of a list or dictionary also define a tuple.

```
1 gang = 'Fred', 'Daphne', 'Velma', 'Shaggy', 'Scooby'
```

Tuples also have one other neat syntactic trick: when you assign a sequence to a tuple of variables, the sequence gets **unpacked** into the variables within the tuple. Unpacking is essentially a compact assignment of values from a sequence to multiple variables.

```
1 # This assignment unpacks the values (right)
2 # into the variables (left)
3 x, y = 3, 4
4
5 # This assignment unpacks a list
6 x, y = [5, 6]
7
8 # This assignment unpacks a range
9 x, y, z = range(3)
10
11 # A common recipe to swap two values uses tuples
12 x, y = y, x
```

Tuples get used quite a bit in Python when small temporary lists or variable unpacking is needed.

## 6.4 Sets

Lists in Python may contain duplicate elements:

```
1 x = [1, 2, 3, 2, 4, 2]
```

But for many problems its useful to keep track of a single instance of a value. In playing the game hang man, you might use a list to represent the guessed letters:

```
1 guessed_letters = ['a', 's']
```

If the player guesses a letter again you might naively append it to the list to get:

```
1 guessed_letters.append('a')
2 guessed_letters → ['a', 's', 'a']
```

But that's not quite right - 'a' is already in the list of guessed letters. Instead of using a list for guessed letters we could use a set:

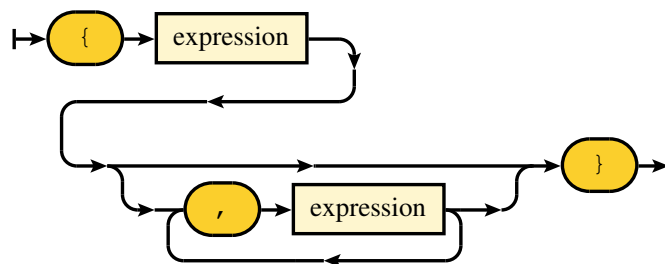
```
1 guessed_letters = {'a', 's'}
```

Sets are a collection of unique items, so adding an item already in the set won't change the set:

```
1 guessed_letters.add('a')
2 guessed_letters → {'a', 's'}
```

The syntax for sets uses the same curly braces as dict but sets are distinguished by not having key value pairs.

#### set literal



Note that literal sets must contain at least one item. To construct an empty set, we can use the `set()` constructor with an empty list:

```
1 empty = set([])
2 empty → {}
```

Much like the keys of a `dict`, sets may only be constructed from hashable (i.e., immutable) values.

Table 64: *Common Set Operations*

Operation	Description
<code>x = {y_1, ..., y_n}</code>	Create a new set with <code>y_1 ... y_n</code> .
<code>set(x)</code>	Create a set from sequence <code>x</code>
<code>y in x</code>	Returns True if <code>y</code> is in the set <code>x</code> Returns False otherwise.
<code>x.add(y)</code>	Add <code>y</code> to the set <code>x</code>
<code>x.discard(y)</code>	Remove <code>y</code> from the set <code>x</code>
<code>x.difference(y)</code>	Returns a set which is the difference of <code>y</code> from <code>x</code>
<code>x.intersection(y)</code>	Returns a set which is the intersection of <code>y</code> with <code>x</code>
<code>x.issubset(y)</code>	Returns True if <code>y</code> is a subset of <code>x</code> . Returns False otherwise.
<code>x.issuperset(y)</code>	Returns True if <code>y</code> is a superset of <code>x</code> . Returns False otherwise.

## Glossary

**dictionary** A data structure that maps unique keys to values.

**immutable** A value that can not change.

**key value pair** The mapping between an object used for lookup, the key, to the stored object, the value.

**list** A data structure that represents an ordered collection of values.

**mutable** A value that can change.

**set** A collection of unordered unique values.

**tuple** An immutable ordered collection of values.

**tuple unpacking** When assignment is used to place values of a sequence into variables within tuple.

**Reading Questions**

- 6.1. Are spacing and newlines within a list or dictionary definition significant?
- 6.2. What kinds of values can be stored in a list?
- 6.3. How do you create an empty list?
- 6.4. Are lists mutable or immutable? What does that mean?
- 6.5. What is meant by the terms key and value?
- 6.6. What kinds of data-types can be used as keys in a dictionary?
- 6.7. What kinds of data-types can be used as values in a dictionary?
- 6.8. What do lists of mixed types often indicate?
- 6.9. How is a dictionary different from a list?
- 6.10. What is a multi-dimensional list?
- 6.11. What does calling `len()` on a dictionary tell you about the dictionary?
- 6.12. What operators are similar in both strings and lists?
- 6.13. What operators are similar in both lists and dictionaries?
- 6.14. You can determine if a value or a key is in a list or a dictionary respectively using the “in” operator. For large datasets which do you think performs “in” faster? Why?
- 6.15. Give three examples of immutable data-types?
- 6.16. What is meant by the word mutable?
- 6.17. How do lists and tuples differ?
- 6.18. What is tuple unpacking?
- 6.19. How is a single member tuple declared?
- 6.20. Is a list within a list is an example of a nested data structure?

## Exercises

### 6-1. Fortune teller

Write a program that defines a function to randomly display a 'Fortune' from a list of quotes. You can make up your own fortunes or grab some from [here](#).

Test code:

```
1 fortune()  
2 fortune()  
3 fortune()
```

Sample run:

```
1 Soon you will be starting a new career.  
2 A refreshing change is in your future.  
3 The wit of a graduate student is like champagne.
```

### 6-2. SMS language expander

Write a function that takes SMS shorthand as an argument and returns the expansion of that shorthand. Use a dictionary to map shorthand to their full meaning. If you can't find a term, return the original string. Your function should handle terms in a case insensitive way. You can find a list of common SMS phrases in Wikipedia's article on [SMS language](#).

Test code:

```
1 print(sms_expander('AFAIK'))  
2 print(sms_expander('BTW'))  
3 print(sms_expander('rotfl'))  
4 print(sms_expander('YOWZERS'))
```

Sample run:

```
1 as far as I know  
2 by the way  
3 rolling on the floor laughing  
4 YOWZERS
```

### 6-3. Pain scale

Pain scales often uses terms like "very mild" and "intense" to describe pain. Write a function which uses a dictionary to translate a human pain rating to a numerical score.

no pain - 0  
very mild - 1  
discomforting - 2  
tolerable - 3

distressing - 4  
very distressing - 5  
intense - 6  
very intense - 7  
utterly horrible - 8  
unbearable - 9  
unimaginable - 10

Test code:

```
1 print(translate_pain("very_mild"))
2 print(translate_pain("intense"))
3 print(translate_pain("unbearable"))
```

Sample run:

```
1 1
2 6
3 9
```

#### 6-4. Chess values II

In Problem [3-3](#) you probably used an if statement to map chess pieces to their common strategic point values. Lets revisit that problem but this time store the values in a dictionary and define a function that when passed a piece name returns its value.

Test code:

```
1 print(chess_value("Rook"))
2 print(chess_value("Pawn"))
3 print(chess_value("Bishop"))
```

Sample run:

```
1 5
2 1
3 3
```

#### 6-5. Roman numerals II

Create a function which takes a number from 0-99 and returns the corresponding Roman numeral (see Problem [3-1](#)). Think about how individual digits contribute to the numeral and how you might **map** Arabic decimals to their Roman counterparts. If your Roman numeral composition skills are rusty, you might consult Wikipedia's article on [Roman numerals](#).

Test code:



```
1 print(roman(7))
2 print(roman(17))
3 print(roman(77))
```

Sample run:

```
1 VII
2 XVII
3 LXXVII
```

### 6-6. Contains vowels

Write a function that takes a single word as its argument and returns True if the string contains a vowel and False otherwise.

Test code:

```
1 print(has_vowel("car"))
2 print(has_vowel("hmm"))
3 print(has_vowel("gypsy"))
```

Sample run:

```
1 True
2 False
3 True
```

### 6-7. Winning at tic-tac-toe

Design a function that takes in a tic-tac-toe board and checks for all possible win conditions. A tic-tac-toe board can be represented as a list of lists see 6.1. If a board contains a winning combination display the appropriate winner.

Test code:

```
1 board = [['X', 'O', 'O'],
2          ['_', 'X', 'O'],
3          ['_', '_', 'X']]
4
5 print(tic_tac_toe_win(board))
```

Sample run:

```
1 True
```

### 6-8. Drop the lowest score

Sometimes teachers will drop the lowest score in calculating a student's grade. Write a function that removes the smallest value from a list and returns the result. Hint: the list doesn't have to come back in the same order!

Test code:

```

1 print(remove_smallest([95, 92, 67, 85]))
2 print(remove_smallest([76, 81, 89, 67]))
3 print(remove_smallest([81, 75, 86, 89]))

```

Sample run:

```

1 [85, 92, 95]
2 [81, 76, 89]
3 [89, 81, 86]

```

### 6-9. Parsing time

Write a function that takes a string in “HH:MM:SS” format and returns a tuple with the integer hours, minutes, and seconds that make up the string. Hint: recall you can use `split` to break apart strings!

Test code:

```

1 print(parse_time("12:00:00"))
2 print(parse_time("3:30:26"))
3 print(parse_time("9:01:47"))

```

Sample run:

```

1 (12, 0, 0)
2 (3, 30, 26)
3 (9, 1, 47)

```

### 6-10. Parsing calendar dates

Write a function that takes a string in “MONTH DAY, YEAR” format and returns a tuple with the month, year and day given as an integer. Hint: you might lookup the month using a dictionary.

Test code:

```

1 print(parse_time("February 2, 1975"))
2 print(parse_time("May 5, 2005"))
3 print(parse_time("August 25, 2014"))

```

Sample run:

```

1 (2, 2, 1975)
2 (5, 5, 2005)
3 (8, 25, 2014)

```

### 6-11. Lights out

In the game “lights out” we are given a 4x4 board of lights. The goal of the game is to turn off all the lights. This is complicated by the fact that toggling a light also toggles its neighbors (i.e., above, below, left, and right). Design a function that toggles a value on a lights out board where

0 is off and 1 is on using an x and y parameter. Your function should return the resulting board.

Test code:

```

1 board = [[0, 0, 0, 0],
2           [0, 1, 0, 0],
3           [0, 1, 1, 0],
4           [0, 0, 0, 0]]
5 print(toggle(board, 1, 1))
6
7 board = [[1, 0, 0, 0],
8           [0, 1, 0, 0],
9           [0, 1, 1, 0],
10          [0, 0, 0, 0]]
11 print(toggle(board, 0, 0))

```

Sample run:

```

1 [[1, 1, 0, 0],
2  [1, 0, 1, 0],
3  [0, 0, 1, 0],
4  [0, 0, 0, 0]]
5
6 [[0, 1, 0, 0],
7  [1, 1, 0, 0],
8  [0, 1, 1, 0],
9  [0, 0, 0, 0]]

```

### 6-12. Matrix multiplication

Define a function that takes in two matrices represented as Python lists and performs matrix multiplication. Do this for 2x2 matrices and then extend it to work with 3x3 matrices.

Test code:

```

1 m1 = [[2,3],[3,2]]
2 m2 = [[4, 5], [6, 5]]
3
4 matrix_multiply(m1, m2)

```

Sample run:

```

1 [[26, 25], [24, 25]]

```

### 6-13. Matrix addition and subtraction

Define functions for adding and subtracting 2x2 matrices that are represented as nested lists.

Test code:

```

1 m1 = [[2,3],[3,2]]
2 m2 = [[4, 5], [6, 5]]
3 m3 = [[1, 2], [3, 4]]
4
5 matrix_add(m1, m2)
6 matrix_subtract(m1, m2)
7 matrix_subtract(m3, m1)

```

Sample run:

```

1 [[6, 8], [9,7]]
2 [[-2, -2],[-3, -3]]
3 [[-1, -1], [0, 2]]

```

#### 6-14. Matrix inverse

Define a function that takes in a matrix represented as a Python list and returns its inverse.

Test code:

```

1 m1 = [[2,3],[3,2]]
2 m2 = [[4, 5], [6, 5]]
3 m3 = [[1, 2], [3, 4]]
4
5 matrix_inverse(m1)
6 matrix_inverse(m2)
7 matrix_inverse(m3)

```

Sample run:

```

1 [[-2/5, 3/5], [3/5, -2/5]]
2 [[-1/2, 1/2],[3/5, -2/5]]
3 [[-2, 1], [3/2, -1/2]]

```

#### 6-15. Affine rotation

Counterclockwise rotation is given by the matrix multiplication:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Define a function that implements counterclockwise rotation given an angle in radians.

Test code:

```

1 rotate([2, 3], 1.6)
2 rotate([2, 3], 3.14)
3 rotate([2, 3], 5)
4 rotate([2, 3], 30)

```

Sample run:

```

1 [-2.06, 2.91]
2 [-2.00, -3.00]
3 [2.49, -2.03]
4 [2.28, -2.50]

```

### 6-16. Affine shear

Shear is a progressive displacement and can be done with respect to the x or y axis in 2D using matrix multiplication:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & k_x \\ k_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Define a function that implements shear given the shear values  $k_x$  and  $k_y$ .

Test code:

```

1 shear([2, 3], 0, 2)
2 shear([2, 3], 2, 0)
3 shear([2, 3], 2, 2)
4 shear([2, 3], .5, 0)
5 shear([2, 3], 3, 9)

```

Sample run:

```

1 [2, 9]
2 [6, 3]
3 [6, 9]
4 [3, 3]
5 [8, 30]

```

## Chapter Notes

The array data-structure is one of the oldest and most fundamental data structures in computing. One can think of an array as simply being a sequence of adjacent addresses in memory. This was a natural data structure used by most of the earliest digital computers. In 1957 FORTRAN became the first high-level language to support multi-dimensional arrays. As languages developed more sophisticated memory management, arrays could be resized by reallocating the memory used to hold the array. When used as a data structure these variable sized arrays are called dynamic arrays. Since this is how Python's list type is implemented, it might be more accurately called a dynamic array.

While many of the exercises in this text have you use lists to create vectors and arrays, more efficient and robust matrices and vectors are available in

`numpy` (available from <http://numpy.org>). In this example we use nested lists to define a `numpy` matrix:

```
1 from numpy import matrix
2
3 m = matrix([[1, 2, 3],
4             [4, 5, 6],
5             [7, 8, 9]])
```

The dictionary data structure used by Python goes by a number of different names. Some languages, including Smalltalk, Objective-C, REALbasic, and Swift use the name dictionary. In Perl and Ruby they are called hashes. In C++ and Java they are called maps. Lua calls them tables. Common Lisp calls them hash tables. PHP calls them associative arrays. And in JavaScript all objects behave like dictionaries so they are often just called objects.

The first language to support dictionaries as part of the language syntax was SNOBOL4 in 1967 [49] but syntactic support for the data structure was popularized by AWK (1977) [47] and Perl (1984) [48].

- [47] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. **The AWK Programming Language**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1987. [isbn: 0-201-07981-X](#).
- [48] Tom Christiansen, brian d foy, and Larry Wall. **Programming Perl**. O'Reilly & Associates, Inc., Sebastopol, CA, 3rd edition, 2000. [isbn: 0596000278](#).
- [49] D. J. Farber, R. E. Griswold, and I. P. Polonsky. SNOBOL, a string manipulation language. **Journal of the ACM**, 11(1):21–30, January 1964. [doi: 10.1145/321203.321207](#).

# 7

## Iteration

*Controlling complexity is the essence of computer programming.*

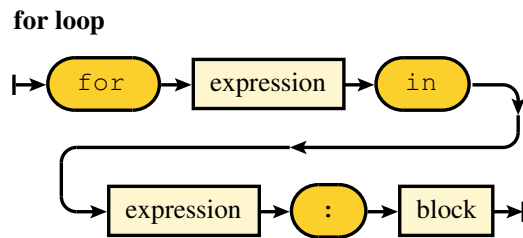
*Brian W. Kernighan*

```
1 overview = ["for_loops",
2             "while_loops",
3             "break_and_continue",
4             "Pattern_Generalization",
5             "Glossary",
6             "Reading_Questions",
7             "Exercises",
8             "Chapter_Notes"]
```

Iteration is a common mechanism for repeatedly evaluating a block of code. Iteration is a powerful tool for solving problems because we can think of the result of one iteration as being the starting point for the next iteration. Thus, iteration is one way to break up big problems into smaller ones that have similar structure. Iteration is also useful for working over different kinds of sequences and objects that are iterable like lists, strings, and ranges of numbers.

### 7.1 for Loops

In Python, a for loop repeats a computation over a sequence of items. As the for loop iterates over each item it assigns the item to one or more iteration variables.



for loops are a real workhorse in Python. Many problems can be solved with iterative approaches and Python provides a number of iterable data structures.

#### *iterating on lists*

Lists are one of the most common iterable data structures. When we use a for-loop with a list, at each iteration the next item in the list is placed in the iteration variable. Consider this example where we iterate over a list of names:

```

1 people = ["Neetha", "Greg", "Michael"]
2 for person in people:
3     print(person + " likes computer science!")

```

Output:

```

Neetha likes computer science!
Greg likes computer science!
Michael likes computer science!

```

When the code inside the for loop is executed the first time, the value of person is “Neetha”. When the code inside the for loop is executed the second time, the value of person is “Greg”. When the code is executed the final time, the value of person is “Greg”.

The code inside a for loop continues to have access to all the variables outside the for loop which can be used or changed from within the for loop. One common pattern is to use one or more variables as an accumulator - something that represents a partial computation that becomes more complete over time.

**Strategy 3 (Accumulation Pattern).** *Use loops with accumulation variables to build up a solution from simpler calculations evaluated over a sequence or condition.*



An accumulation pattern is used in this code that sums exam grades. The `exam_total` variable is the accumulator. It begins at zero and then at each iteration is adds one exam grade to the running total.

```

1 # We have three grades we'd like to sum
2 exam_grades = [85.0, 92.0, 100.0]
3
4 exam_total = 0.0
5 for grade in exam_grades:
6     exam_total += grade
7
8 exam_total → 277.0
9
10 exam_average = exam_total / len(exam_grades)
11 exam_average → 92.33333333333333

```

In fact, we can generalize this pattern using pseudocode:

```

1 initialize accumulator
2 loop until we calculate final result
3     update accumulator

```

This pattern works in many situation. Consider this code that creates an acronym one letter at time by iterating over a list of words.

```

1 # We have three grades we'd like to sum
2 words = ["Frequently", "Asked", "Questions"]
3 acronym = ""
4 for word in words:
5     acronym += word[0].upper() "."
6
7 acronym → 'F.A.Q.'

```

#### *iterating on strings*

Another interesting iterable data-type is the string. When you iterate over a string the iteration variable takes on the value of each character in the string. The next example uses this to count the vowels in a word:

```

1 word = "supercalifragilisticexpialidocious"
2

```

```

3 vowel_count = 0
4 for letter in word:
5     if letter in 'aeiou':
6         vowel_count += 1
7
8 vowel_count → 16

```

Python has methods for changing the case of a string to upper or lower case. Using iteration we can create a custom method that inverts the case of a string!

```

1 def invert_case(s):
2     ns = ""
3     for c in s:
4         if c.isalpha() and c.isupper():
5             c = c.lower()
6         elif c.isalpha():
7             c = c.upper()
8         ns += c
9     return ns
10
11 invert_case("Scooby_Doo") → 'sCOOBY_dOO'

```

We can also iterate over substrings using `split()`.

```

1 def reverse_words(sentence):
2     words = []
3     for word in sentence:
4         words.insert(0, word)
5     return "_".join(words)
6
7 reverse_words("Good_work_Scoob") →
8 "Scoob_work_Good"

```

Did you spot the accumulation pattern we used? In this example we built up the result by adding elements to the beginning of the list instead of the end!

#### *iterating on ranges*

for loops can also be used to repeat a block of code a specific number of times. This is accomplished with a range object, which creates a sequence of integers. The parameters to the range function are described in Table 71.

Table 71: Range Parameters

Mode	Description
<code>range(i)</code>	Create a sequence of integers from 0 to <code>i</code>
<code>range(i, j)</code>	Create a sequence of integers from <code>i</code> to <code>j</code>
<code>range(i, j, k)</code>	Create a sequence of integers from <code>i</code> to <code>j</code> stepping <code>k</code> units

In the simplest case, we can use `for` and `range` to repeat a block of code a fixed number of times:

```
1 number_sum = 0
2 for number in range(10):
3     number_sum += number
4 number_sum → 45
```

But using the start, stop and step parameters the `range` function can generate many linear sequences:

```
1 for number in range(0, 10, 2):
2     print(number, end=" ")
```

Output:

```
0 2 4 6 8
```

These sequences can count down as well as up:

```
1 for number in range(10, 0, -2):
2     print(number, end=" ")
```

Output:

```
10 8 6 4 2
```

We can also create more complicated sequences by using the `range` values to parameterize a more complicated calculation. This loop calculates `n` factorial:

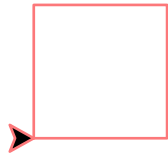
```
1 def factorial(n):
2     n_fact = 1
3     for i in range(1, n+1):
4         n_fact = n_fact * i
```

```
5     return n_face
6
7 factorial(9) → 362880
```

range is a general mechanism for repetition. In this example that draws a square, the variable `i` isn't used and that's ok.

```
1 import turtle
2
3 for i in range(4):
4     turtle.fd(50)
5     turtle.lt(90)
```

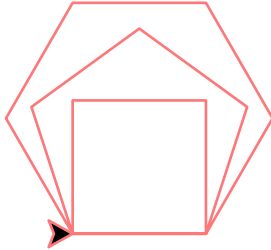
Output:



We can use a function to generalize the last example such that it draws an **n**-sided polygon:

```
1 import turtle
2
3 def draw_ngon(n):
4     for i in range(n):
5         turtle.fd(50)
6         turtle.lt(360.0/n)
7
8 draw_ngon(4)
9 draw_ngon(5)
10 draw_ngon(6)
```

Output:



### *nesting loops*

All control structures in Python can be nested. It's useful to think of the nested levels as representing dimensions (like columns and rows). We can print a multiplication table by nesting ranged for loops within each other.

```

1 for left in range(1, 10):
2     for right in range(1, 10):
3         print("{:2}".format(left * right), end=" ")
4     print()
5
6 1  2  3  4  5  6  7  8  9
7 2  4  6  8 10 12 14 16 18
8 3  6  9 12 15 18 21 24 27
9 4  8 12 16 20 24 28 32 36
10 5 10 15 20 25 30 35 40 45
11 6 12 18 24 30 36 42 48 54
12 7 14 21 28 35 42 49 56 63
13 8 16 24 32 40 48 56 64 72
14 9 18 27 36 45 54 63 72 81

```

The inner loop prints each column of a single row, while the outer loop evaluates each inner loop repeatedly - printing each row in turn.

In this example we use the same nested structure to draw squares. The inner loop draws a single square and the outer loop repeats the operation.

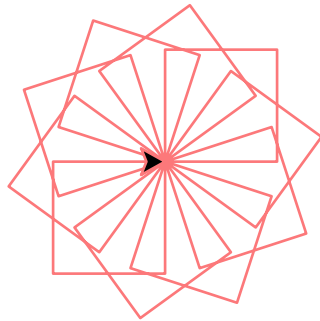
```

1 import turtle
2
3 nsquares = 12

```

```
4  sidelen = 42
5  for i in range(nsquares):
6      for j in range(4):
7          turtle.fd(sidelen)
8          turtle.lt(90)
9      turtle.lt(360.0 / nsquares)
```

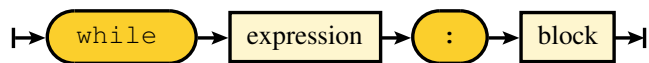
Output:



## 7.2 while Loops

The for statement is powerful but it doesn't let us easily loop on an arbitrary condition. For that we need the while loop.

### while loop



A while loop executes continuously until a condition is met. In the next examples we consider several common use cases.

*search loops*

*calculating loops*

Consider the problem of computing a square root. One common technique for this problem is to employ Newton's method of successive approximations. Whenever we have a guess,  $y$ , for the value of the square root of a number,  $x$ , we can get a better guess by averaging  $y$  with  $x/y$ . Using this technique we can compute the square root of 2 as follows with an initial guess of 1:

Table 72: *Newton's Method*

Guess	Quotient	Average
1	$(2 / 1) = 2$	$((2 + 1) / 2) = 1.5$
1.5	$(2 / 1.5) = 1.3333$	$((1.3333 + 1.5) / 2) = 1.4167$
1.4167	$(2 / 1.4167) = 1.4118$	$((1.4167 + 1.4118) / 2) = 1.4142$
1.4142	...	...

Continuing this process, we obtain better and better approximations to the square root. But our termination condition might be better stated in terms of the accuracy we want the approximation to achieve rather than the number of steps it should take.

```

1 def average(x, y):
2     return (x + y) / 2.0
3
4 def sqrt(x, guess=1):
5     while abs(guess ** 2 - x) >= 0.001:
6         guess = average(guess, x / guess)
7     return guess
8
9 sqrt(5.0) → 2.23606889564

```

The while loop in the last example terminate's because Newton's method converges on the solution. Sometimes we choose termination conditions that can't be met. Sometimes this is on purpose (for example, when we want to use break and continue to control the iteration) and sometimes it's unintentional (i.e., a bug). When loops are unable to terminate we say that we have created an **infinite loop**.

Consider this simple example of an infinite loop:

```

1 while True:
2     print('Computer Science Rocks!')

```

Output:

```

Computer Science Rocks!
Computer Science Rocks!
Computer Science Rocks!
Computer Science Rocks!
...

```

The only way to terminate an infinite loop is to terminate the process that is running the loop. In general, we want to build functions and software systems that terminate in well defined ways and we must consciously avoid designing inadvertent infinite loops.

#### *event loops*

An event loop is a construct that waits for an event and then dispatches or processes the event. Event loops are very common in user interface toolkits (although the actual loop may be hidden by the framework). A very simple event loop that does a simple validation of input from the console might look like this:

```

1 user_input = ""
2 while user_input not in ["a", "b", "c"]:
3     user_input = input("Please type a, b, or c: ")
4 print("You chose '{}'".format(a))

```

Output:

```

Please type a, b, or c: d↵
Please type a, b, or c: e↵
Please type a, b, or c: a↵
You chose 'a'.

```

### 7.3 break and continue

Both while loops and for loops may terminate under certain conditions but its sometimes useful to terminate a loop within the logic of the loop. The



break construct can be used to terminate a loop immediately. Once broken the program flow continues after the while loop.

```
1 i = 2
2 while(i < 100):
3     j = 2
4     while j <= (i/j):
5         if not i % j:
6             break
7         j = j + 1
8     if (j > i/j) :
9         print (str(i) + " is prime")
10    i = i + 1
11 print("done")
```

continue doesn't terminate the loop but it does terminate the current iteration - passing control back to the beginning of the loop.

#### 7.4 Pattern Generalization

### Glossary

**infinite loop** A loop that will never terminate because its termination condition can not or will not be satisfied.

**iteration** A technique used to repeat a computational process.

**nested loop** A loop that is embedded within another loop.

### Reading Questions

- 7.1. What do we call a loop that will never terminate?
- 7.2. What do we call loop embedded inside another loop?
- 7.3. What is a technique used to repeat a computational process?
- 7.4. How can an infinite loop be terminated?
- 7.5. Which type of statement can we use to exit a loop whose iteration condition can't be met?

- 7.6. Can `while` loops repeat on an arbitrary condition? If so, how?
- 7.7. What is a construct that waits for an event and then dispatches or processes the event?
- 7.8. Can all control structures in Python can be nested?
- 7.9. What can we use to execute a for loop a specific number of times?
- 7.10. What is the difference between `break` and `continue`?
- 7.11. Do accumulation variables have to be numeric?
- 7.12. What do the `i` and `j` indicate in the range parameter `range(i, j)`?
- 7.13. What is the purpose of iteration?
- 7.14. Consider the following statement:

```
1 for num in range(2, 10):  
2     print (num)
```

What is the value of `num` on the 4th iteration of the loop?

- 7.15. Consider the following code:

```
1 name_list = ['Ricky', 'Fred', 'Lucy', 'Ethel']  
2  
3 for name in name_list:  
4     print(name)
```

What is the value of `name` on the 2nd iteration of the loop?

- 7.16. What does a `continue` statement do?
- 7.17. What is each pass through a loop called?
- 7.18. When does a `while` loop terminate?
- 7.19. Consider the following code:

```
1 i = 0  
2  
3 while i <= 10:  
4     print(i)
```

When will the `while` loop exit?

- 7.20. What does a `break` statement do?

## Exercises

### 7-1. Pig Latin translator II

Problem 3-14 asked you to write a Pig Latin translator for individual words. In this problem you should build on this solution to develop a function that can translate whole sentences into Pig Latin.

Test code:

```
1 print(pig_latin("Today is a nice day."))
2 print(pig_latin("I can not wait until lunch."))
3 print(pig_latin("It is about time."))
```

Sample run:

```
1 Odaytay isyay ayay icenay ayday.
2 Iyay ancay otnay aitway untilyay unchlay.
3 Ityay isyay aboutyay imetay.
```

### 7-2. Ubbi dubbi

Ubbi Dubbi is a language game similar to Pig Latin where “ub” is added before each vowel sound in a word. Write a program that takes a sentence as input and outputs the sentence in Ubbi Dubbi.

Test code:

```
1 print(ubbi_dubbi("Hi, how are you?"))
2 print(ubbi_dubbi("What is your major?"))
3 print(ubbi_dubbi("Computer Science is a good choice."))
```

Sample run:

```
1 Hubi, hubow ubarube yubou?
2 Whubat ubis yubour mubajubor?
3 Cubompubutuber Scubiencube ubis uba gubood chuboicube.
```

### 7-3. Sum of cubes

Write a program that finds the sum of the cubes of the first  $n$  natural numbers.

Test code:

```
1 print(sum_of_cubes(10))
2 print(sum_of_cubes(20))
3 print(sum_of_cubes(100))
```

Sample run:

```
1 3025
2 44100
3 25502500
```

**7-4. ROT13**

ROT13 is a simple substitution cypher that replaces each letter with another letter 13 letters later in the alphabet. If a substitution letter would go past Z it wraps back to A.

Test code:

```
1 print(rot13("ROT13 is an example of the Caesar
2             cipher."))
3 print(rot13("This cypher is fairly easy to crack."))
4 print(rot13(rot13("It was sufficient in the time of
5             the Romans.")))
```

Sample run:

```
1 EBG13 vf na rknzcyr bs gur Pnrfne pvcure.
2 Guvf plcure vf snveyl rnfl gb penpx.
3 Vg jnf fhssvpvrag va gur gvzr bs gur Ebznaf.
```

**7-5. There was an old lady who swallowed a fly!**

Write a program that uses for loops to display the verses of “There was an old lady who swallowed a fly!” This song has a **cumulative song** structure, meaning that each verse is longer than the one before due to progressive additions. The full lyrics to the song are available at [online](#). (Hint: consider using lists to store the song elements).

**7-6. Converting for-loops**

Take the program you wrote for Problem 7-5 and modify it to use a **while** loop instead of a **for** loop.

**7-7. Magic square validator**

A Magic Square is a grid of numbers where the figures in each vertical, horizontal, and diagonal row add up to the same value. Write a program that can check whether the below square is in fact a magic square.

11	18	25	2	9
10	12	19	21	3
4	6	13	20	22
23	5	7	14	16
17	24	1	8	15

**7-8. Pyramids**

Using Python’s turtle package make a program that draws a triangle and displays it in the following positions: normal, upside down, left, right. Have your program iterate through displaying each position.

**7-9. Checkerboard**

Write a program that prints out a checker board pattern in an  $n \times n$  size taken from user input.

**7-10. Rock-paper-scissors III**

Extend the rock-paper-scissors game from Problem 3-19 to play multiple rounds. The first player who gets best two out of three wins the game.

**7-11. Fibonacci sequence**

Write a program that defines a function `fib_sequence()` which displays the Fibonacci sequence numbers between 0 and n.

Test code:

```
1 fib_sequence(25)
```

Sample run:

```
1 0 1 1 2 3 5 8 13 21 34 55 8 144,
2 233, 377, 610, 987, 1597, 2584, 4181,6765,
3 10946, 17711, 28657, 46368
```

**7-12. High-low game**

Write a program that creates a guessing game where the player is prompted to guess a number between 1 and 1000. If the player's guess is too low display "{guess} is too low. Try Again.", do the same if it is too high, then let the player try again. When the player guesses correctly congratulate them and display how many guesses it took them to win. This program should use a while loop, and if-elif-else statements. Use Python's random module to generate your number between 1 and 1000.

Here is a sample run:

```
1 Guess a number between 1 and 1000: 546↵
2 546 is Too high. Try again.
3
4 Guess a number between 1 and 1000: 314↵
5 314 is too low. Try again.
6
7 Guess a number between 1 and 1000: 402↵
8
9 Congratulations! You won in 3 guesses.
```

**7-13. Maclaurin cosine estimator**

The Maclaurin series expansion of  $\cos(x)$  is given by,

$$\cos(x) = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k}}{(2k)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!}$$

This formula is valid for all real values of x.

Create a program that estimates the Maclaurin series expansion of cosine using iteration. Your program should ask a user for user input, and use that input as the parameter of the cosine estimation function you create.

**7-14.  $\pi$  estimation**

$\pi$  is the ratio of the circumference to the diameter of a circle. You can estimate  $\pi$  stochastically by drawing a circle inscribed in a square and then randomly picking points in the square. If the points are random and uniformly distributed, you can keep track of which points land just in the circle,  $P_c$ , and which land anywhere in the square,  $P_s$ , using the equation for a circle (see Problem 5-11). Given that the circle has area  $A_c = \pi r^2$  and the square has area  $A_s = (2r)^2$ , we expect the ratio  $P_c$  to  $P_s$  should be the same as  $A_c$  to  $A_s$ . Solving for  $\pi$  you should get,  $\pi = 4A_c/A_s$ .

Write a function that takes the number of points as input and returns an estimation of  $\pi$ . What happens as you use more points?

**7-15. Monty Hall problem**

In the Monty Hall problem you are given a choice between three doors. Behind one door is a car and behind the other doors are goats. After selecting a door, the host opens one of the doors you did not select which contains a goat. He then asks you if you want to switch to the other unopened door?

This problem has stumped many a statistician - is it better to stick with the same door or switch? Model the problem in Python with a function that returns if it was better to switch (a `bool`) for a single instance of the game. Then use iteration to run your function 1000 times and count how many times out of 1000 it was better to switch.

**7-16. Morse code translator**

Create a function that can take in a string in the English language and translate it to morse code using the following Morse code dictionary:

```

1 mcode = {
2     'A': '.-', 'B': '-...',
3     'C': '-.-.', 'D': '-..',
4     'E': '.', 'F': '..-..',
5     'G': '--.', 'H': '....',
6     'I': '..', 'J': '.---',
7     'K': '-.-', 'L': '.-..',
8     'M': '--', 'N': '-.',
9     'O': '---', 'P': '-.-..',
10    'Q': '--.-', 'R': '.-..',
11    'S': '...', 'T': '-',
12    'U': '...-', 'V': '...-',
13    'W': '---', 'X': '-...-',
14    'Y': '-.-.-', 'Z': '----',
15    '0': '-----', '1': '.-----',
16    '2': '..-----', '3': '...-----',
17    '4': '....-----', '5': '.....-'
```

```

18     '6': '-....', '7': '--...',
19     '8': '-----', '9': '-----'}

```

Your program should also be able to deal with lowercase letters without adding elements to the dictionary.

### 7-17. Morse code visualization

Using Python's turtle graphics package, write a program that can visually display the morse code you generate. Use the time module to delay the time between each sequence displayed.

### 7-18. Mastermind

The Mastermind game is a famous code breaking game invented in 1970 by Mordecai Meirowitz. If you are unfamiliar with the game you can read about it here [http://en.wikipedia.org/wiki/Mastermind\\_\(board\\_game\)](http://en.wikipedia.org/wiki/Mastermind_(board_game)). Write a program that implements the Mastermind game by asking a user to chose four color choices per turn, and then gives feedback regarding correct color and position. Allow the user eight turns to guess.

Sample run (one round):

```

1  The colors are:
2  Red, Yellow, Green, Blue,
3  Purple, and Orange
4
5  Guess a sequence of four colors
6  seperated by spaces: P R G Y
7
8  Result:
9  2 correct color and position
10 2 correct color only

```

### 7-19. Scrabble word score calculator

Create a program that asks a user to input a word and then calculates the word's scrabble word score. The value of letters in the scrabble game can be found at <http://www.scrabblefinder.com/scrabble-letter-values/>.

Test code:

```

1 print(word_score("Hello"))
2 print(word_score("Equation"))
3 print(word_score("Zebra"))
4 print(word_score("Test"))
5 print(word_score("Zilch"))

```

Sample run:

1	8
2	17
3	26
4	4
5	19

## Chapter Notes

Iteration is one of the oldest and most fundamental ideas in computing and dates back to antiquity. Euclid's algorithm, for example, uses iteration to find the greatest common divisor of two integers. The algorithm appears in Euclid's "Elements" which dates to approximately 300 BC [52].

In most early programs iteration was accomplished with a goto statement. FORTRAN introduced a do loop in 1957 and ALGOL 58 and 60 introduced while loops and for loops.

It should be noted that Python's for loop is what is often called a foreach loop - a specialized for loop construct that considers individual elements in a sequence. Foreach like constructs could be created with macros in LISP and Scheme and using messages in Smalltalk. The Bourne shell (1977) was the first language where foreach was explicitly made part of the language syntax using a for-in construct similar to Python [50, 51].

- [50] Steve Bourne. Bourne shell. **Version 7 UNIX Programmer's Manual**, 1979. [link](#).
- [51] Howard Dahdah. The A-Z of programming languages: Bourne shell, or sh. **Computerworld Australia**, 2009. [link](#).
- [52] Jeffrey Shallit. Origins of the analysis of the Euclidean algorithm. **Historia Mathematica**, 21(4):401 – 419, 1994. [doi: http://dx.doi.org/10.1006/hmat.1994.1031](https://doi.org/10.1006/hmat.1994.1031).



## 8

# Input and Output

*On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.*

*Charles Babbage*

```
1 overview = ["Files",
2             "Serialization",
3             "JSON_*",
4             "XML_*",
5             "Glossary",
6             "Reading_Questions",
7             "Exercises"
8             "Chapter_Notes"]
```

Until now most of the input we have considered has come directly from the user or has been coded as part of our program. In real systems there are many different sources for input and output including events from user interfaces, network communications, hardware sensors, databases and file systems.

In this chapter we will discuss files and common data formats for transporting structured data.

## 8.1 Files

In many programming languages files can be accessed with a proxy object called a **file handle**. The file handle is a reference to the file and provides methods for reading and writing the file.

In Python we create a file handle (or file object) using the built-in function `open()`. The first argument to the function is the name of the file and the second is called the access mode. Some of the most common access modes are summarized in Table 81. Additional mode suffixes 'b' and '+' can be used to specify the file is binary or should allow simultaneous read and write access.

Table 81: *Common File Access Modes*

Mode	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

Once we have obtained a file object, we can perform operations on the file with methods associated with that object. Some common file methods are summarized in Table 82.

Table 82: *Common File Methods*

Mode	Description
<code>x.close()</code>	flushes any unwritten information and closes the file object, after which no more writing can be done.
<code>x.write(y)</code>	writes the string <code>y</code> to an open file
<code>x.read()</code>	Reads as much of the file as possible.
<code>x.readline()</code>	Read one line of a file

Thus to write a file named “hello.txt” to disk with the contents “Hello world!”, we could write:

```
1 hello_file = open("hello.txt", "w")
2 hello_file.write("Hello world!")
3 hello_file.close()
```

Then to read the file back into a string we could write:

```
1 hello_file = open("hello.txt", "r")
2 hello_string = hello_file.read()
3 hello_file.close()
```

## 8.2 Serialization

Serialization is the process of turning objects in memory into data that can be transported somewhere else (a file, network, etc.). Serialization is a way to save the state of an object such that it can be reconstructed at a later time.

In Python, serialization is usually accomplished using the pickle module. Pickle can do a lot of advanced things but it’s also pretty simple to use. Consider this code that takes a dictionary and saves it to disk:

```
1 import pickle
2
3 color = {"Fred": "blue", "Daphne": "purple"}
4 pickle.dump(color, open("colors.p", "wb"))
```

The resulting file, colors.p, isn’t designed to be human readable but it can be easily reconstituted in Python:

```
1 import pickle
2
3 color = pickle.load(open("colors.p", "rb"))
```

Pickled files may contain Python code in them so you should never evaluate a pickle file you don’t trust.

Serialization is often used to allow the state of our program to persist between different runs.

```
1 import pickle
2 import random
3
```

```

4 print("Guess my number game 1.0")
5 command = input("Type 'new' or 'load': ")
6
7 if command == 'load':
8     game = pickle.load(open("gamesave.p", "rb"))
9 else:
10    game = {}
11    game["name"] = input("\nType your name: ")
12    game["number"] = random.randint(0, 10)
13
14 while True:
15    guess = input("\nGuess 0-10 or 'quit': ")
16    if guess == 'quit':
17        pickle.dump(game, open("gamesave.p", "wb"))
18        break
19    elif guess == int(game["number"]):
20        print(game["name"] + ", that's correct!")
21        break
22    else:
23        print(game["name"] + ", try again.")

```

1 Output:

```

2
3 Guess my number game 1.0
4 Type 'new' or 'load': new↵
5
6 Type your name: James
7 Guess 0-10 or 'quit': 2↵
8 James, try again.
9 Guess 0-10 or 'quit': quit↵

```

1 Output (from my second run):

```

2
3 Guess my number game 1.0
4 Type 'new' or 'load': load↵
5
6 Guess 0-10 or 'quit': 5↵

```

```
7 James, try again.  
8 Guess 0-10 or 'quit': 3↔  
9 James, that's correct.
```

The game itself isn't very impressive but the fact that we were able to preserve the state between two different runs with only a couple of lines of code is powerful.

The trick to using pickle is making sure that all of the state you want to preserve is accessible from a single variable.

### 8.3 JSON \*

While pickle is very powerful it also has a few draw backs. Since pickled data may contain executable code, pickle should not be used to read untrusted data. Pickle is also not human readable nor is it easy to read pickled data outside of Python.

A number of data formats exist that have better readability and portability than pickle at the expense of less tight integration with Python.

JSON is one such format and while it's name (JavaScript Object Notation) might suggest it is tightly bound to JavaScript, that's hardly the case. JSON uses the JavaScript notation for representing objects, lists and primitive types as a data representation language. This is a particularly good fit for Python because Python's list and dictionary literals are very similar to JavaScript's list and object literals. Conversion to and from JSON works much like Pickle:

```
1 import json  
2  
3 color = {"Fred": "blue", "Daphne": "purple"}  
4  
5 color_json = json.dumps(color)
```

Unlike pickle, JSON intends to be human readable and parameters can be used to effect what the JSON output looks like:

```
1 json.dumps(color, indent=2)
```

The output then maps back into Python objects:

```
1 color = json.loads(color_json)
```

**Reading Questions**

- 8.1. In many programming languages files can be accessed with an object called what?
- 8.2. How do you write a statement that will open a file called "hello.txt" for reading in binary?
- 8.3. Can JSON can be mapped into Python objects?
- 8.4. What is the default file access mode when opening a file?
- 8.5. In Python we create a file handle (or file object) using which built-in function?
- 8.6. How much of a file does the read() file method read?
- 8.7. What does the close() file method do?
- 8.8. What is serialization?
- 8.9. What is the correct way to close file handle 'hello\_file'?
- 8.10. Where is the file pointer positioned when a file is opened for appending with the 'a' mode?
- 8.11. Which statement will open a file called "hello.txt" for writing?
- 8.12. What can the JSON data format be used to represent?
- 8.13. Are Python's list and dictionary literals similar to JavaScript's list and object literals?
- 8.14. Why should you never evaluate a pickle file you don't trust?
- 8.15. What is a file handle and what does it allow us to do?
- 8.16. When is the close() file method necessary?
- 8.17. What arguments are passed to the open() method?
- 8.18. Using the mode suffix '+' does what?
- 8.19. Using the mode suffix 'b' specifies the file is what?
- 8.20. A file object is synonymous with file handle?

## Exercises

### 8-1. Fortune teller II

Extend the fortune program you wrote in Problem 6-1 to read the fortunes from a file where fortunes have been written one per line.

### 8-2. Acronym definitions II

Extend the acronym lookup program you wrote in Problem X to read acronyms from a file where each line contains an acronym, a space, and the acronym expansion.

Test code:

```
1 acronym_lookup('As_soon_as_possible')
2 acronym_lookup('Back_at_computer')
3 acronym_lookup('I_want_to_go_to_lunch')
```

Sample run:

```
1 ASAP
2 BAC
3 Sorry couldn't find an acronym for your phrase
```

### 8-3. War and peace

Project Gutenberg has been creating plain text versions of classic books readable by humans and computers since 1971. Go to <http://www.gutenberg.org/cache/epub/2600/pg2600.txt> and save the plaintext version of Leo Tolstoy's novel **War and Peace** as `war_and_peace.txt`. Write a program that will read in the `.txt` file and count the occurrences of the words 'war' and 'peace' in the novel. Make sure to account for upper and lower case words and print out how many times each word occurs.

### 8-4. Common words

Using the `war_and_piece.txt` file you previously saved, write a program that gives you the ten most common words that occur in the text. Display each word and its number of occurrences.

### 8-5. Lines and words

Using the `war_and_piece.txt` file you previously saved, write a program that gives you a count of the total number of words as well as the total number of lines in the text.

### 8-6. Pig Latin translator III

Extend the Pig Latin translator from Problem 7-1 to work on whole files.

### 8-7. ROT13 II

Extend the ROT13 cypher from Problem 7-4 to work on whole files.

**8-8. Magic square validator III**

Extend the magic square validator from Problem 7-7 to work on magic squares given in comma separated value files.

**8-9. Checkerboard II**

Extend the checkerboard printer from Problem 7-9 to write the checkerboard to a file instead of stdout.

**8-10. Rock-paper-scissors IV**

Extend the rock-paper-scissors game from Problem 7-10 to keep track of the number of games the player wins and how many games the computer wins over multiple runs of the program using pickle.

**8-11. Mastermind II**

Extend the mastermind game from Problem 7-18 to allow the player to quit at any time. When the player quits save the state of the game using pickle such that they can continue their game the next time they play.

**8-12. Task list**

Write a program that implements a simple to-do list. Your program should prompt the user to a) list all tasks, b) add a task, c) remove a task, or d) quit. You'll need to do some list manipulation and you will need to store your task list to a file when the user quits and automatically load it when the program starts again.

**8-13. Matrix multiplication II**

It's very common to multiply many matrices together one after another. This is done in computer graphics to do affine transformations and is done in robotics to computer reference frames. Write a program that reads a csv file with an arbitrary number of 3x3 matrices, multiplies the matrixes together, and prints the result (see Problem 6-12).

**8-14. Saving data with JSON**

Use a Python dictionary to describe one of the classes you are taking. Use the following attributes as keys in your dictionary: 'id', type int; 'title' type string; 'section', type int; 'description', type string; 'enrolled', type boolean; 'passing', type boolean; 'tags', type a dictionary of strings. Once you have created your dictionary open a new file in write mode and use the `json.dump()` function to serialize your data. What does the content of your file look like?

**8-15. PPM image reader**

A PPM image file has a header followed by image data. Write a program that reads in an ASCII format PPM image file and then returns the images width and height.



A PPM image header has the following format:

```
P6 – magic number
# comment– comment lines begin with #
# another comment – any number of comment lines
200 300– image width & height
255 – max color value
```

A sample run might look like:

Test code:

```
1 image = open('my_image.ppm')
2 read_ppm(image)
```

Sample run:

```
1 Image width: 512
2 Image height: 512
```

## 8-16. Facebook stats

Ever wonder how Facebook keeps track of all your information? Well, using JSON is one way they have of serializing your data. Visit [developers.facebook.com/tools](https://developers.facebook.com/tools), choose GET and enter 'me' in the search field to see a JSON representation of your user information. Now model your Facebook stats using a Python dictionary with the information you obtained from Facebook or the following fields: id, birthday, first\_name, last\_name, gender, link, location\_name, timezone. Print out your dictionary in JSON indented 4 spaces with sorted keys.

Test code:

```
1 print(facebook_stats(my_info))
```

Sample run:

```
1 {
2     "birthday": "07/21/1990",
3     "email": "joe_smith@gmail.com",
4     "first_name": "Joe",
5     "gender": "male",
6     "id": "1619285896",
7     "last_name": "Smith",
8     "link": "http://www.facebook.com/1619285896",
9     "locale": "en_US",
10    "location": {
11        "id": "119818622580173",
12        "name": "Flagstaff, Arizona"
13    },
14    "name": "Joe Smith",
15    "timezone": -7,
```

```

16     "updated_time": "2014-04-07T02:06:29+0000",
17     "verified": true
18 }

```

### 8-17. Lexical diversity

Lexical diversity is a measure of the number of different words used in a text. The works of William Shakespeare are known to show off his huge vocabulary, but as [Matt Daniels](#) discovered, Rap artists have huge vocabularies as well. Create a program that reads in a .txt file and prints out the number of unique words in the text, and the text's lexical diversity. Lexical diversity can be found by dividing the total number of words in the text by the total number of unique words. You can get the complete works of William Shakespeare at [Project Gutenberg](#) to test your program.

Test code:

```

1 text = 'complete_shakespeare.txt'
2 lexical_diversity(text)

```

Sample run:

```

1 Total unique words: 24009
2 Lexical Diversity: 38.83

```

### 8-18. Letter frequency

Frequency analysis is often used to break encrypted messages. Create a program with a function `letter_freq()` that takes in a .txt file and builds a frequency listing of the characters contained in it. Represent the frequency listing as a Python dictionary.

### 8-19. Command-line spellchecker

Create a command line program that reads in a .txt file and checks the spelling of all the words in the text. You will need a dictionary of correctly spelled words to check against. Download the .txt file located at <http://www.puzzlers.org/pub/wordlists/unixdict.txt> and use this as your dictionary. If a word is spelled incorrectly print out the message "The word <word> may be incorrectly spelled."

### 8-20. Population statistics

U.S. population statistics are commonly stored as .csv (comma separated values) files. Download 2013 population projections at [census.gov](http://census.gov) and save as a .csv file. Write a program that reads in the .csv file and prints out each state name along with its estimated population as of July 1, 2013.

Test code:

```

1 file = 'NST-EST2013-01.csv'
2 pop_stats(file)

```

Sample run:

```
1 Alabama: 4,833,722
2 Alaska: 735,132
3 Arizona: 6,626,624
4 Arkansas: 2,959,373
5 California: 38,332,521
6 ...
7 .....
```

## Chapter Notes

Most languages derive their file handling semantics from C. This is evident even in Python where the file modes 'r', 'w' and 'rw' directly correspond to those in C [25].

One of the earliest cross-platform data formats was CSV which was supported in FORTRAN compilers as early as 1967. CSV and TSV remain common formats for tabular data.

XML is a derivative of SGML, the same family of markup from which HTML is descended. SGML's lineage in academia and industry goes back to IBM's Generalized Markup Language developed in 1969. The first working draft for XML was produced in 1996 and became a W3C recommendation in 1998. XML intends to be easy for both humans and machines to read but in practice it can be quite verbose. XML has strong support for checking the validity and well-formedness against a reference specification and is independent of any programming language or specific application. For readers interested in learning more about XML Harold and Means' "XML in a Nutshell" [55] provides a good introduction.

JSON was originally derived from JavaScript's literal object and list notation by Douglas Crockford in 2001. Although derived from JavaScript, the JSON syntax is similar to Python, Ruby, and Swift and is relatively easy to map into array and associative array containers available in most modern programming languages. This close correspondence between the syntactic structure of JSON documents and the language structures used to access and manipulate that data has made JSON a popular alternative to other more abstract data formats. JSON is described formally in ECMA-404 and RFC 7159 [54, 53]. A description of JSON and a JSON parser is also provided in Crockford's "JavaScript: The Good Parts" [7].

While language independent formats like XML and JSON can be used to save program data, saving program data in such a way that it reconstitutes the types, objects, and methods in a program is necessarily language specific. Serialization facilities are available in a host of languages including Python, Ruby, Perl, PHP, Java, .NET, and Objective-C. But because serialization is typically platform dependent, software developers often choose more general data formats when interoperability is of concern.

[53] Tim Bray. RFC-7159: The javascript object notation (JSON) data interchange format, 2014. [link](#).

[54] ECMA. ECMA-404: The JSON data interchange format, 2013. [link](#).

[55] E.R. Harold and W.S. Means. **XML in a Nutshell**. O'Reilly Media, 2004. [isbn: 9781449379049](#).

## 9

# Classes and Objects

*The data represent an abstraction of reality in the sense that certain properties and characteristics of the real objects are ignored because they are peripheral and irrelevant to the particular problem. An abstraction is thereby also a simplification of the facts.*

*Niklaus Wirth*

```
1 overview = ["Classes",
2             "Methods",
3             "Fields",
4             "Encapsulation",
5             "Inheritance",
6             "Polymorphism",
7             "Identity",
8             "Class_Members_*",
9             "Properties_*",
10            "Glossary",
11            "Reading_Questions",
12            "Exercises"
13            "Chapter_Notes"]
```

Object-oriented programming is a paradigm where related state and behavior are organized into classes that define objects. In Python the jump to object-oriented programming is more of a hop than a leap. The reason is that all the types we've used so far are examples of objects! While we've been using objects

in every chapter; now we will start designing our own objects and describe some of the organizing principles of object-oriented design.

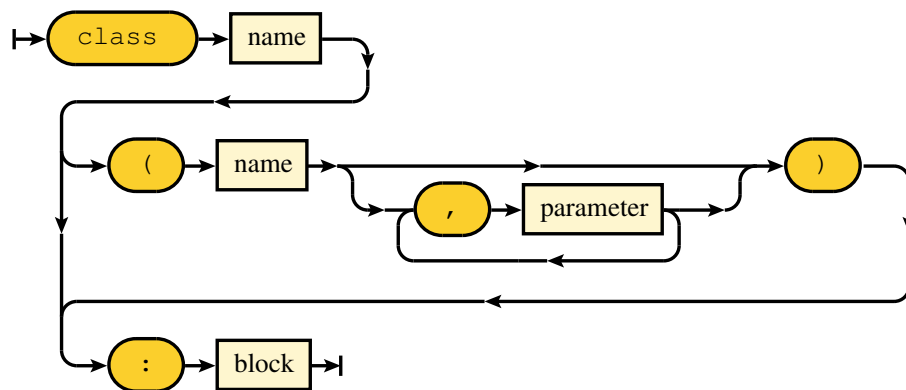
## 9.1 Classes

An object is a collection of state and behavior. In Python, almost everything is an object - strings, numbers, Booleans, functions, and modules are all examples of objects.

We design new kinds of objects with classes. A **class** is kind of like a blue print for an object. The class describes what state and behavior should make an object but it is not the object itself.

We define classes with the keyword `class`. This is followed by the class name, a colon, and then the indented body of the class. The class body contains the methods and field definitions that define the object.

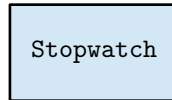
### class declaration



The practice of object-oriented programming spans multiple steps of the four step software development process we introduced in Chapter 1. As an organizational and abstraction tool it falls into Step 2 - planning. But as a mechanism to write code it falls into Step 3 - implementation and testing.

Fortunately there are a number of ways to do object-oriented design without committing to specific implementation details. One popular approach to designing classes involves diagramming objects using something called Unified Modeling Language (UML) class diagrams.

In the next sections we will define a Stopwatch. In UML the simplest class description is just a box that gives the name of the class:



The corresponding implementation in Python looks like this:

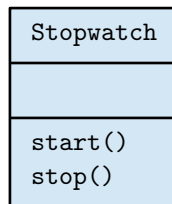
```
1 class Stopwatch:
2     pass
```

Class names in Python follow a convention where each word starts with a capital and is followed by lower case letters.

## 9.2 Methods

Methods provide objects with behavior. In this example, our `Stopwatch` needs to start and stop a timer.

In UML diagrams, behavior can be added by subdividing the class box into three boxes representing the class name, fields, and behavior respectively:



The corresponding Python code looks like this:

```
1 class StopWatch:
2
3     def start(self):
4         pass
5
6     def stop(self):
7         pass
```

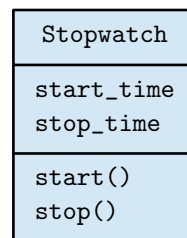
In Python, method definitions look just like function definitions except that they have an extra first parameter that usually represents the object the method is acting on. By convention it is called `self`.

The one thing that UML class diagrams don't tell us is how to implement these methods. Class diagrams provide a way to design class structure and

relationships which make these diagrams especially useful as planning tools. This then becomes the plan for creating working code. Since we aren't quite ready to add that working code we use the special Python keyword `pass` as a syntactic place holder.

### 9.3 Fields

Most objects are combination of behavior and state. Objects maintain state in **fields** - variables associated with the object.



In Python we typically initialize fields in a special method called a constructor. The constructor is responsible for initializing the object and is always named `__init__()`. For the most part a constructor works like any other method.

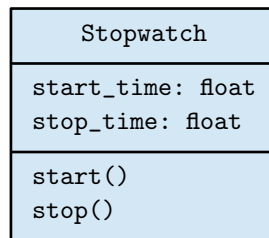
```
1 from time import time
2
3 class Stopwatch:
4
5     def __init__(self):
6         self.start_time = 0
7         self.stop_time = 0
8
9     def start(self):
10        self.start_time = time()
11
12    def stop(self):
13        self.stop_time = time()
```

Fields represent the state of the object and can be accessed or changed from



other methods. In the `Stopwatch` example we use fields to keep track of when the `start()` and `stop()` methods get called<sup>1</sup>.

While Python doesn't have a mechanism for annotating the types of fields, UML does. UML owes this to its history as a tool often used in designing classes for C++ and Java which are statically typed languages. While it might initially seem like a mismatch to annotate types for a dynamically typed language, it can be very helpful when making design decision concerning how different components fit together. Type annotation in UML is accomplished using a colon that follows the named entity.



Type annotations may also follow parameter names and functions.

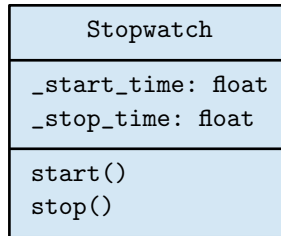
## 9.4 Encapsulation

Encapsulation is an object oriented principle where implementation details are hidden and programmers are encouraged or required to only use public fields or methods. This allows the hidden (also called private) implementation to change over time without having to worry about breaking code that depends on it.

In the `Stopwatch` example, the start and stop times are implementation details of the `Stopwatch` - it's not our intention that code outside this class change these variables directly. In Python, private members are distinguished by starting with a single underscore. Public members have no special prefix. We can represent this change in UML:

---

1. Python's `time()` function gives the number of seconds that have passed since January 1, 1970 (also called the epoch). For more date and time functions see the Python Documentation section on [time](#).



Python has three levels of visibility:

1. public, intended for access anywhere,
2. private, intended for restricted access (usually within the same class or module), and
3. class private, intended for access only within the same class.

Class private members begin with a two underscore prefix and you've already seen one example - constructors. Another example is the `__str__()` method which is used by `str()` and `print()` to stringify objects which aren't already strings.

```

1 from time import time
2
3 class Stopwatch:
4
5     def __init__(self):
6         self.start_time = 0
7         self.stop_time = 0
8
9     def __str__(self):
10        return str(self.stop_time -
11                   self.start_time)
12
13    def start(self):
14        self.start_time = time()
15
16    def stop(self):
17        self.stop_time = time()

```

Now if we `str()` our stopwatch, we will get the elapsed time:

```
1 timer = Stopwatch()
2 timer.start()
3 # wait a couple of seconds
4 timer.stop()
5 str(timer) → 2.1013
```

Class private members are often used for methods that have special meaning in Python but are less common in garden variety Python code<sup>2</sup>.

One important question you might ask is how do we decide which members should be private and which public? A good rule of thumb is that fields should be private unless you have a compelling reason not to make them private. Methods will typically be public unless the method represents some internal functionality (i.e., a helper function).

UML has a special annotation for marking the visibility of fields and methods using prefix symbols '+', '-', '#', and '~' but these marks are really designed for C++ and Java which have very different visibility modifiers<sup>3</sup>.

In Python, visibility is by convention and is apparent from the name of the field or method. The fact that we have made `_start_time` private doesn't prevent other objects from accessing the field but it serves as a message to other developers that the field is part of the internal implementation.

## 9.5 Inheritance

When we design objects with classes we often encounter a design problem where different classes are often very similar. One way to reuse code is to take advantage of behavioral and state similarities through a process called inheritance.

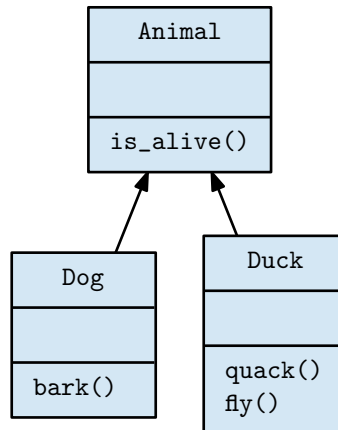
Inheritance is a way of defining a new class, called a **subclass** or **child class**, based on an existing class, called the **parent class**. By default the child class inherits the fields and methods of its parent. We can then **specialize** the child class with completely new fields and methods or even methods that **override**

---

2. Class private variables have special semantics that involve name mangling for intraclass calls, which are outside the scope of this text but which you can read about in the Python Documentation section on [Private Variables](#).

3. While Python's public members correspond closely with Java and C++ public members, it gets murkier after that. Python's private access is really more like Java's package access. Python's concept of class private is sort of like Java's private access and Python has no clear analog to Java's protected. And that's just one interpretation.

existing behavior. In UML we represent inheritance with an arrow that connects a child class to its parent:



In this example, Duck and Dog inherit from Animal. Because of the inheritance each gets a copy of the method `is_alive()`. Each then provides specializations like `make_sound()` or `growl()`. A collection of classes and their inheritance relationships is called an **inheritance hierarchy**.

In general, inheritance is an **additive** process. You can use it to add (or change) methods and state but not remove them. By pushing shared behavior up the hierarchy and specialized behavior to the bottom we can reduce potentially replicated code in similar classes.

Inheritance is easily represented in Python. After declaring a class we use parenthesis after the class to list one or more parent classes:

```
1 class Animal:
2     def is_alive(self):
3         return True
4
5 class Duck(Animal):
6     def make_sound(self):
7         print("Quack-quack")
8
9 class Dog(Animal):
10    def make_sound(self):
11        print("Woof-woof")
```

## 9.6 Polymorphism

When a method is called on an object in Python the behavior depends on the underlying type of the object. When the behavior of a method depends on the underlying type of object we say the behavior is **polymorphic**.

Many object-oriented languages link polymorphism to inheritance. Python is a bit different. Methods are looked up and called completely at runtime from the underlying type of an object. Thus, all methods are automatically polymorphic - it's something you don't even have to think about in Python. Consider this code:

```
1 q1 = Duck()
2 q2 = Dog()
3 q1.make_sound()
4 q2.make_sound()
```

`q1` and `q2` will make very different sounds because of their underlying types. Consider this example where the type is less obvious:

```
1 animals = [Duck(), Dog(), Duck(), Dog()]
2
3 for animal in animals:
4     animal.make_sound()
```

How does Python know which `make_sound()` method to call for `animal`? Each time it encounters `animal.make_sound()` it looks up the method appropriate for that type which exposes the polymorphic behavior.

## 9.7 Comparing Objects

When we compare strings and numbers we are usually concerned about comparing their value. We could, for example, construct a string in two very different ways and would expect that they have equal value:

```
1 s1 = 'cab'
2 s2 = ''.join(['c', 'a', 'b'])
3 s1 == s2 → True
```

But are `s1` and `s2` the same object? We can determine if two objects are truly the same with the `is` operator (first introduced in Section 3.6):

```
1 s1 = 'cab'
2 s2 = ''.join(['c', 'a', 'b'])
```

```
3 s1 is s2 → False
```

When two objects evaluate as equal using the `is` operator they are in fact the same object. When they evaluate as equal using the `==` operator it just means that according to the comparison operation for that type they have the same value - but they are not necessarily the same object.

One special case worth mentioning is `None`. While `None` represents the absence of an object reference it is in fact a kind of object itself and in Python there is only one. So if we want to check if a variable holds `None` we use the `is` operator.

```
1 s1 = 'cab'
2 s1 is None → False
3 s1 is not None → True
```

Another way we can think of comparing objects is through their inheritance tree. In Python this can be accomplished with the `isinstance()` method which returns `True` if an object's type is a class or subclass of the specified type:

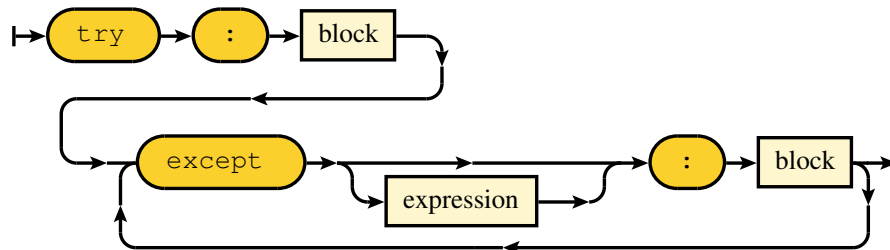
```
1 a1 = Animal()
2 a2 = Beaver()
3 c1 = Car()
4 isinstance(a1, Animal) → True
5 isinstance(a2, Animal) → True
6 isinstance(c1, Animal) → False
```

Writing code that depends on `isinstance()` may not be very robust or portable. Because Python is a very dynamic language, the inheritance hierarchy is a fairly poor way to reason about the capability of objects.

## 9.8 Error Handling

Exceptions are a special kind of error that change the normal flow of execution such the closest error handling code in the function call stack is called. If no error handling code is found, the program halts.

Exception handling has two important components. The `try` block is the code you would like to “try” to execute. If the code in the `try` block raises an exception it immediately stops executing and the `except` block that handles the error is evaluated.

**try statement**

See Table 91 for a listing of some common built-in exception types. For a complete list of built-in exceptions see the Python Documentation section on [Built-in Exceptions](#).

Table 91: *Some Common Exceptions*

Name	Description
<code>AssertionError</code>	Raised when an <code>assert</code> fails.
<code>AttributeError</code>	Raised when an attribute reference or assignment fails.
<code>IndexError</code>	Raised when a sequence subscript is out of range.
<code>KeyError</code>	Raised when a dictionary key is not found.
<code>TypeError</code>	Raised when an operation or function is applied to an inappropriate type.

Many of the common tasks we perform in Python raise exceptions. Consider this innocuous looking code, which accesses a dictionary key called “name”:

```
1 name = user["name"]
```

If “name” isn’t in the `user` dictionary, the program will halt with a `KeyError` exception. One way to prevent this error is to check if the key exists first:

```
1 if "name" in user:
2     name = user["name"]
3 else:
4     name = "default"
```

This error handling approach is sometimes called Look Before You Leap (LBYL). But we could also execute the code and then handle the error after the fact:

```

1  try:
2      name = user["name"]
3  except KeyError:
4      name = "default"

```

Exceptions enable an error handling approach that is sometimes called Easier to Ask Forgiveness than Permission (EAFP). Both approaches have their merits but exception handling can sometimes help us avoid tedious error checking. Consider the problem of placing an X randomly on a checkerboard represented as a list of lists. If we practice LBYL, we might write:

```

1  i = random.randint(0, len(board))
2  j = random.randint(0, len(board))
3  if i > 0 and i < len(board) and \
4      j > 0 and j < len(board):
5      board[i][j] = 'X'
6  else:
7      print("Illegal move")

```

We do the error checks to prevent the X from being placed outside the bounds of the board. Using EAFP, the same effect is achieved without checking if *i* and *j* are correctly bounded:

```

1  i = random.randint(0, len(board))
2  j = random.randint(0, len(board))
3  try:
4      board[i][j] = 'X'
5  except IndexError:
6      print("Illegal move")

```

Besides handling exceptions, we can also generate them. This is accomplished with the `raise` statement followed by the exception to raise. One common use case for exception generation is checking the type of a parameter:

```

1  def reverse_string(a_string):
2      if isinstance(a_string, str):
3          raise TypeError('expected string')
4      return a_string[::-1]

```



Then we could use this in code that calls `reverse_string()`:

```
1 try:
2     print(reverse_string(7892)):
3 except TypeError:
4     print("Should have known better")
```

## 9.9 Class Members \*

While we have focused on fields and methods associated with objects, we can also associate fields and methods with classes. Variables defined in the class become class variables and are accessed as properties of the class:

```
1 class NoiseMaker:
2
3     count = 0
4
5     def __init__(self):
6         NoiseMaker.count += 1
7
8 n1 = NoiseMaker()
9 n2 = NoiseMaker()
10 print(NoiseMaker.count)
```

Instead of being associated with an individual instance, `count` is associated with the `NoiseMaker` class. One way to think about class variables is that there is a single `count` that all `NoiseMaker` objects share. Members associated with a class are a bit like global variables but have the advantage that they are part of class namespace. This can prevent global naming conflicts and can reduce your reliance of the `global` statement.

**Principle 4** (Avoid Global Variables). *Code that depends on global variables often represents poor modularity. Try to parameterize your solution in terms of instance variables and where necessary consider using class variables.*

Creating class methods usually involves using a special bit of decorator syntax. Decorators are special functions that rewrite Python code. Writing them is out of the scope of this text, but using them is easy. In this example we use the `@classmethod` decorator to turn `reset_count()` into a class method.

```
1 class NoiseMaker:
2
3     count = 0
4
5     def __init__(self):
6         NoiseMaker.count += 1
7
8     @classmethod
9     def reset_count(cls):
10        cls.count = 0
11
12 n1 = NoiseMaker()
13 n2 = NoiseMaker()
14 NoiseMaker.reset_count()
15 print(NoiseMaker.count)
```

### 9.10 Properties \*

A common design pattern involves guarding a private variable with functions that get and set the value (called getters and setters):

```
1 class Point:
2
3     def __init__(self):
4         self._x = 0
5         self._y = 0
6
7     def get_x(self):
8         return self._x
9
10    def get_y(self):
11        return self._y
12
13    def set_x(self, x):
14        self._x = x
15
16    def set_y(self, y):
```

```
17         self._y = y
```

Using getters and setters designed like this is pretty clumsy, so Python provides a syntax that makes getters and setters look like ordinary fields:

```
1 class Point:
2
3     def __init__(self):
4         self._x = 0
5         self._y = 0
6
7     @property
8     def x(self):
9         return self._x
10
11    @property
12    def y(self):
13        return self._y
14
15    @x.setter
16    def x(self, x):
17        self._x = x
18
19    @y.setter
20    def y(self, y):
21        self._y = y
```

Once again we are using special decorators that re-write our code so that when we use the Point class the getters and setters get called when we access the field:

```
1 p1 = Point()
2 p1.x = 3.0
3 p1.y = 5.0
```

While it looks like we are directly setting fields in the last example, Python is really calling the getter and setter methods we set up.

Unfortunately getters and setters often get abused. In this example we would have been better off not using getters or setters at all because they don't provide any added functionality over using ordinary fields! Consider using Python's

properties when computations need to be performed when fields in the class change or when you want to provide pseudo-properties that don't really exist.

### Glossary

**access** An annotation that specified the under what context a method or field should be available.

**class** A structure used like a blue print to define the methods and fields that make up an object.

**constructor** A special method used to initialize an object.

**encapsulation** A design principle where the internal state and behavior of an object is hidden.

**field** A variable associated with an object.

**inheritance** An object-oriented mechanism where a class can reuse functions and fields defined in a super class.

**method** A function associated with an object.

**polymorphism** A way of calling methods such that the method called is based on the underlying type of the object.

**private** An access type that indicates a feature should be hidden from external objects.

**property** A property is sometimes used as a synonym for a field, but in Python a property is a method wrapped to act like a field.

**public** An access type that indicates a feature should be visible to external objects.

**self** The name of the variable typically used to represent the object that a method acts on.

**subclass** A class that specialized a parent class through inheritance.

### Reading Questions

- 9.1. What is the relationship between a class and an object?
- 9.2. How does UML fit into our four step problem solving process?
- 9.3. How do we tell if two references point to the same object?
- 9.4. Name some features or advantages of object-oriented programming as compared to procedural programming.

- 9.5. How are fields declared in a class?
- 9.6. How are methods declared in a class?
- 9.7. How can object-oriented programming positively effect code reuse?
- 9.8. How can encapsulation positively improve our software design?
- 9.9. What is polymorphism?
- 9.10. How are public and private relationships expressed in Python?
- 9.11. What is a variable associated with an object called?
- 9.12. What word refers to an access type that indicates a feature should be hidden from external objects?
- 9.13. What does the word 'self' refer to?
- 9.14. What is a method?
- 9.15. What do we call a collection of classes and their inheritance relationships?
- 9.16. When should fields be private?
- 9.17. Does a field become part of the state of an object?
- 9.18. How is a private member represented in UML?
- 9.19. What do getter and setter functions do?
- 9.20. How can inheritance help us reuse code?

## Exercises

### 9-1. Spheres

Draw the UML for a class designed to represent a sphere. Your class should include methods for calculating the surface area and volume of the sphere. Once your UML is complete, implement your design in working Python code.

### 9-2. Cubes

Draw the UML for a class designed to represent a cube. Your class should include methods for calculating the surface area and volume of the cube. Once your UML is complete, implement your design in working Python code.

**9-3. Lock digit**

Draw the UML for a class designed to represent a lock digit that can be set to a value between 0 and 9. Your class should have a constructor that sets the locked value, a method that changes the current value, and a method that opens the lock. The open method is successful it should return `True`; otherwise it should return `False`.

**9-4. Lock composition**

Use composition to compose three lock digits from Problem 9-3 into a three digit lock. Your lock should have a constructor that accepts a 3 digit code for the lock, a method that changes the current value, and a method that opens the lock. Draw the UML for your lock and then implement your design in working Python code.

**9-5. Lock composition II**

Extend your solution for Problem 9-4 to build an n-digit lock where n is specified implicitly when a lock code is passed into the constructor for your lock. Draw the UML for your lock and then implement your design in working Python code.

**9-6. Playing cards**

Draw the UML for a class designed to represent a playing card. Your class should have methods for returning the suit, rank (the number or title of the card), Blackjack value, and a `__str__` method. Once your UML is complete, implement your design in working Python code.

**9-7. Card deck**

Draw the UML for a class designed to represent a deck of cards. Your class should use the card class you developed in Problem 9-6. Your deck class should have a method to initialize the deck (the constructor) with 52 cards, shuffle the deck, a method to deal a card by removing it from the deck and returning it, and a method to add a card to the bottom of the deck.

**9-8. SMS language expander II**

In Problem 6-2 you probably used `LBYL` to check if an acronym was in the dictionary before returning the expansion. Revise your solution to use `EAFP` and exception handling.

**9-9. Parsing time II**

In Problem 6-9 you might not have considered invalid input. Revise your solution to raise an error for invalid times.

**9-10. Parsing calendar dates II**

In Problem 6-9 you might not have considered invalid input. Revise your solution to raise an error for invalid dates.

**9-11. Matrix class I**

In Problems [6-13](#), [6-12](#), and [6-14](#) you created functions that work on matrices. Incorporate these methods into a Matrix class which supports addition, subtraction, multiplication, and inversion. You can even make these operation work with the operators `+`, `-`, and `*` by using carefully named methods. See the Python Documentation section on [Emulating numeric types](#).

**9-12. Matrix class II**

Extend Problem [9-11](#) with class methods that create rotation, scale, and skew matrix instances. See Problems [6-15](#) and [6-16](#) for the definition of these operations.

**9-13. Vector class**

Create a vector class which can be multiplied by your Matrix class from problem [9-11](#) to produce new vectors.

**9-14. Lights out II**

In Problem [6-11](#) we described a way of playing the game lights out using zeroes and ones to represent the state of the lights. Lets create a more complete version of the game using OO design. Using UML, design a light class, and a board class which can toggle lights, check for a win condition, display the board, and initialize the board to a random state. Note that just randomly assigning lots on and off for the initial state can create an unwinnable game! Instead, randomly select lights to toggle when you set up the board<sup>4</sup>.

**9-15. Match 3**

Use UML to design an implementation of match-3 on an 8x8 board. Your board will need to be able to keep track of a score, swap elements, and fill cleared areas with new values.

**Chapter Notes**

The notion of modeling problems with objects arose from work done at MIT and within the AI community in the late 1950s. The design of LISP in particular supported problems modeled with LISP atoms and attributes.

The first language that supported object-oriented programming as a language supported feature was Simula 67. Simula introduced classes, instances, subclasses, and virtual method based polymorphism [[60](#)].

---

4. For an extensive list of resources related to the lights out game see [[56](#)] and [[61](#)].

The term object-oriented programming was coined in the 1970s to describe the pervasive use of objects and message passing enabled by Alan Kay's Smalltalk which first appeared in 1971. Smalltalk advocated a design where all values are objects and software systems are reflective [59, 58].

Throughout the 70s many perceived object-oriented programming as inefficient and impractical. In reaction, Bjarne Stroustrup began developing what would become C++ in 1979. C++ supported a kind of object-oriented programming strongly influenced by Simula that was efficient and could co-exist with non-object-oriented C code [62].

Brad Cox and Tom Love also envisioned adding objects to C but used Smalltalk's message passing metaphor to develop Objective-C in 1986 [57].

By the 1990s object-oriented programming had become a staple in language design.

- [56] Margherita Barile. Lights out puzzle. **From MathWorld – A Wolfram Web Resource, created by Eric W. Weisstein.** [link](#).
- [57] Brad J. Cox and Andrew Novobilski. **Object-Oriented Programming; An Evolutionary Approach.** Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2nd edition, 1991. [isbn: 0201548348](#).
- [58] Adele Goldberg. **Smalltalk-80: The Interactive Programming Environment.** Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1984. [isbn: 0-201-11372-4](#).
- [59] Adele Goldberg and David Robson. **Smalltalk-80: The Language and Its Implementation.** Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1983. [isbn: 0-201-11371-6](#).
- [60] Jan Rune Holmevik. Compiling Simula: A historical study of technological genesis. **IEEE Annals in the History of Computing**, 16(4):25–37, 12 1994. [link](#).
- [61] Jaap Scherphuis. Lights out, 2013. [link](#).
- [62] Bjarne Stroustrup. **The Design and Evolution of C++.** Pearson Education, 1994. [isbn: 9788131716083](#).



## 10

# Testing

*Beware of bugs in the above code; I have only proved it correct, not tried it.*

*Donald Knuth*

*Testing reveals the presence, but not the absence of bugs.*

*Dijkstra*

```
1 overview = ["Fail_Early",
2             "Contracts",
3             "Logging",
4             "Unit_Testing",
5             "Equivalence_Classes",
6             "Regression_Testing",
7             "Test_Driven_Development",
8             "Glossary",
9             "Reading_Questions",
10            "Exercises"
11            "Chapter_Notes"]
```

An oft-repeated story asserts that the term “bug” was coined in 1947 when a Harvard Mark II Computer operator traced an error to a moth that had been trapped in a relay. The operator carefully removed the moth and taped it to a log book with the caption “first actual case of bug being found.” While this story

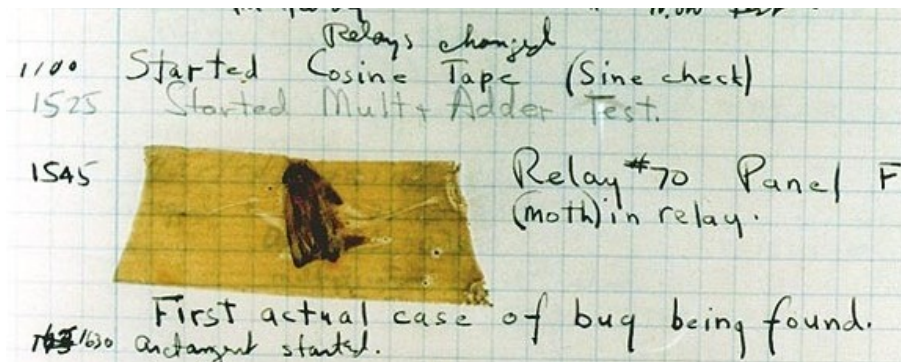


Figure 101: An operator notes "First actual case of bug being found" beneath the remains of a moth taped to a logbook.

is true, the word bug is much older<sup>1</sup>. Thomas Edison recounts bugs arising from intuition and describes them as “little faults and difficulties.” Bugs have been part of the engineering lexicon for decades and aren’t unique to computers in any sense. In the context of software development, a **bug** (also called a software **defect**) is any error or mistake that causes a system to produce incorrect results or have unintentional behavior.

When people write software they often make mistakes. Testing is a practice that intends to find mistakes by identifying incorrect behavior.

Sometimes bugs occur because of typographical errors. A developer may make a syntax mistake because of a misplaced brace or period. A developer might also create a program that works but has incorrect behavior by accidentally using the wrong variable name. Copy and paste errors are particularly common, where developers copy existing code to a new location but forgets to update some of the identifier names to reflect the local context. But there are also more substantial mistakes where a developer arrives at a solution to the problem that fundamentally doesn’t work. These kinds of mistakes may require large blocks of code to be rewritten entirely to express a new approach.

While software defects exist in any non-trivial piece of software, critical bugs are often responsible for lost revenue, data, and in some cases loss of life. In 1983, Atomic Energy of Canada Limited released one of the first software controlled radiation therapy devices, the Therac-25. Between 1985 and 1987

1. This was one of Grace Hopper’s favorite stories from her time working with the Harvard Mark II, although she points out she wasn’t actually present when it happened. The actual bug is now held by the Smithsonian [63].

this device was involved in six accidents that gave patients roughly 100 times the intended dose of targeted treatment. Three of the six patients died due to the error. While the machines worked correctly for hundreds or thousands of hours, certain conditions would cause internal parameters to reset to unknown values while the correct parameters were erroneously displayed to the operator. The Therac-25 incidents are among the most famous examples of software control failures and illustrate poor design and a lack of rigorous testing.

In this Chapter we will discuss several techniques for testing software as a way to be more confident in the correctness of the code that we write. One thing that we will emphasize is that testing shouldn't be an afterthought - it should be something we actively think about as we design and implement software.

## 10.1 Fail Early

As we design software we sometimes recognize that the code we write only works under certain conditions. We might, for example, write a program that draws a circle in a window on our computer. To accomplish this we might call a function to initialize a window and then a second function to draw the circle:

```
1 window = new_window()
2 draw_circle(window)
```

But what if the function to initialize the window failed and returned `None` instead of a window reference? If a value isn't what we expect then the software system is in an **inconsistent state**.

One approach to handling inconsistent state is to ignore the inconsistency or attempt to fix the inconsistency. In this example, if the `draw_circle()` function detects the window isn't valid it might just silently do nothing (ignoring the problem) or it might try to create its own window (trying to fix the problem).

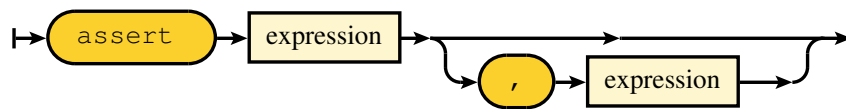
Inconsistent state is the result of a software defect - by ignoring the state or attempting to fix the inconsistency when it's detected, we may be simply differing or promulgating erroneous behavior. When we encounter inconsistent state - there often is no "correct" action that can be taken.

**Principle 5 (Fail Early).** *When an inconsistent state is detected, the program or task should not continue running in an inconsistent state. Instead the program should halt and identify the internal inconsistency.*

The effect of aggressively failing early is that errors can be detected with temporal or lexical proximity to the bug that caused the inconsistency.

We can do this in Python with something called an assertion. An **assertion** evaluates an expression and if it is `False` then it raises an error which will halt the program if not caught.

#### assertion



The error does more than simply stop the program. It also informs the user what assertion failed and where it failed. If we change the last example to use an assertion it might read:

```

1 window = new_window()
2 assert window is not None
3 draw_circle(window)

```

If the window isn't initialized correctly the program will not continue running. Instead we get an error message that helps us understand what error occurred.

**Principle 6** (Fail On Your Own Terms). *Think about critical points or mechanisms by which your software could fail and use assertions to fail with rich information about the failure. Actively avoid conditions that allow the program to fail in ways that don't provide information about the failure.*

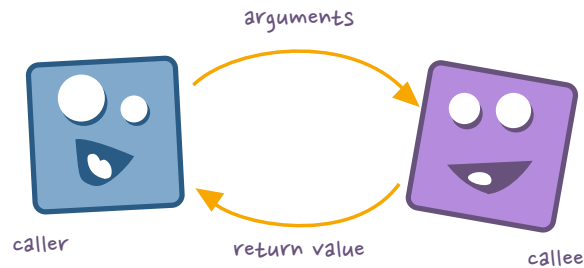
The best software developers are often obsessed with failure. When a failure happens unexpectedly finding the root of the failure can be very difficult but if we can force our software to fail on our own terms, then we give ourselves the power and the tools to identify the conditions that lead to the failure.

## 10.2 Contracts

While the fail early principle is easy to understand, how do we detect inconsistent state and where do we test it? Contractual design suggests that we think about software as being something like a business contract.

When a home buyer has a house built, the expectations are described in a contract. The home buyer has certain expectations of the builder (e.g., they use certain materials, follow a certain design, etc.) and the builder has certain expectations of the buyer (e.g., they have money, rights to the property, etc.).

We can think of program interactions as having a similar sort of relationship. A calling function is a bit like the home buyer (a service consumer) and the called function is a bit like the builder (a service provider):



Of course other artifacts of state and behaviors besides arguments and resources could also be part of this “exchange.” The main point is that functions have expectations about the relationship they have with other functions which we can think of as contractual - if these contractual obligations aren’t met then the state of the program is inconsistent.

**Principle 7** (Think Contractually). *When you design a function, think about the conditions or environment required for the function to work correctly and what constitutes a correct result. Make these relationships part of your design.*

Most functions we write are really “partial functions” - functions that are not well defined across all possible arguments or program state. If we don’t make the assumptions part of a function’s design, it’s easy for a function to yield incorrect results or cause the program to fail.

Contractual design advocates that we articulate the contractual relationships between functions using three different kinds of contracts:

1. **preconditions** - relationships that must hold before a function is executed,
2. **postconditions** - relationships that must hold after a function is executed, and
3. **invariants** - relationships that must hold while a function is executed.

Consider this function, which returns a the first letter of a string after it has been capitalized:

```
1 def first_upper(a_string):  
2     return a_string[0].upper()
```

In order for `first_upper()` to work correctly, a number of assumptions are made about the parameter `a_string`. And when we call `first_upper()`, we have expectations about the result. These assumptions and expectations can be converted into contracts:

- precondition: `a_string` is a `str`
- precondition: `a_string` is not empty
- postcondition: the result is a `str`
- postcondition: the result has a length of one
- postcondition: the result is an uppercase letter

We can then use assertions to implement the contracts:

```
1 def first_upper(a_string):
2     # preconditions:
3     assert isinstance(a_string, str)
4     assert len(a_string) > 0
5
6     result = a_string[0].upper()
7
8     # postconditions
9     assert isinstance(result, str)
10    assert len(result) == 1
11    assert result == result.upper()
12    return result
```

Contracts are much more than a way of adding consistency tests to code – they shape the way that you write code by organizing method relationships around a testable contractual metaphor. The tests can improve the reliability of your code and readability of your code by providing programmatically enforced interface specifications.

### 10.3 Logging

While contracts allow us to create constraints on interactions within a program, some defects are difficult to find without a richer context of the conditions that led to the defect.

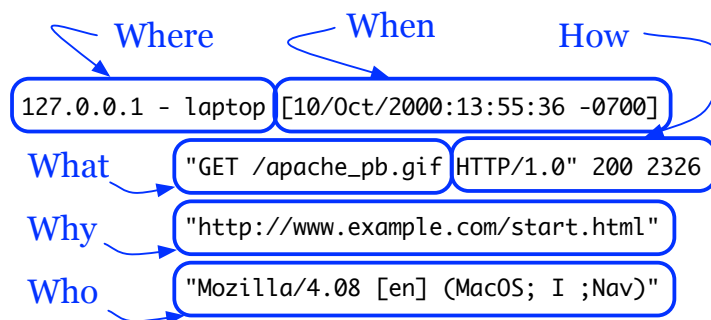
A log is a human readable listing of events which occurred during the execution of a program. Logging intends to report information and answer many of the **questions of circumstance** Cicero raised in antiquity (similar to the 5 W's of Journalism). The questions of circumstance that we ask in logging include:

- What happened?
- Where did it happen?
- When did it happen?
- How important was it?

Logs are important artifacts in computing. Logs are used in web servers to record who visits a website, the web browser used, the page the user visited and many other statistics. Logs are also used by the operating system to record what programs have been executed, who has logged into the system, etc. Log entries are simply lines of text:

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700]
  "GET /apache_pb.gif HTTP/1.0" 200 2326
  "http://www.example.com/start.html"
  "Mozilla/4.08 [en] (Win98; I ;Nav)"
```

At first it may seem a bit cryptic but it compactly answers Cicero's questions of circumstance:



The log entry tells a story. Someone using the Mozilla web browser at address 127.0.0.1 retrieved an image file named "apache\_pb.gif" because it was attached to the start page of a website at <http://www.example.com>. They were able to do this using the HTTP/1.0 protocol, their request was a success, and was made up of 2326 bytes.

This particular format is a common one for web server logs and is called the Common Log Format (CLF). It's an example of what a good log entry should do – tell a complete story in a single line. Most web servers have logs with thousands or millions of entries. It's not practical to have log entries depend on each other or span multiple lines. Logs should be informative and concise.

**Principle 8** (Use Logs to Tell a Story). *Good program logs tell a story about the situations your software encounters and the actions the software takes in response. Each line should tell a small story and multiple lines should tell a bigger story.*

Logs can be used to create statistics about how software has been utilized (e.g., how many people visited a page on a web site) or to understand how software has failed (e.g., finding failed login attempts in an operating system log).

The worst kinds of logging doesn't tell us useful information. Even seasoned programmers are often guilty of inserting lines like this to test reachability in their code:

```
1 print("got_ here")
```

What's wrong with this code? At best it considers one or two of Cicero's circumstances and it doesn't even identify them very well. This kind of logging is adhoc, unorganized and if we leave these statements in we will forget why they are even there. Good logging frameworks can be turned on and off, target different output mediums (e.g., text files, email, web services), and have rich capabilities for controlling what log entries look like.

Python has a flexible and easy to use logging system available with a simple import:

```
1 import logging
```

The default log object has methods for different kinds of log entries that indicate the importance or kind of entry being described. These are summarized in Table 101.

Table 101: *Log Categories*

Mode	Description
debug	Information that is useful when debugging a problem but otherwise may not be particularly useful.



Mode	Description
info	Information that is provided when processes are working correctly.
warning	Information which points to unusual system state.
error	The application is unable to proceed normally because of reasons indicated.
critical	A very serious error has happened which will likely cause the system to terminate or become unresponsive.

This code fragment illustrates how each category can be used as a method call to identify a condition:

```
1 logging.debug("bytes_read_from_file")
2 logging.info("file_processed")
3 logging.warning("out_of_space_for_file")
4 logging.error("missing_metadata")
5 logging.critical("file_contents_missing")
```

Additional information could be added to the message (such as the specific file name, space constraints, or metadata that was missing) with the format method described in Section 4.7.

You can also set the level at which messages should be displayed, and the format of log entries using methods described in the Python Documentation section on [logging](#). In the next example we use logging as we prompt the user for some information before we play a game:

```
1 import logging
2
3 # Add additional logging info:
4 logging.basicConfig(
5     "[% (filename)s: %(lineno)s] " + \
6     "% (message)s")
7
8 logging.info("requesting_user_data")
9
10 response = input("Would you like to play a game?")
```

```
11 if response not in ["yes", "no"]:  
12     logging.error("got bad response, '{}'".  
13                 format(response))  
14  
15 name = input("What is your name? ")  
16 if name != "Falken":  
17     logging.warn("user, {}, is playing".  
18                 format(name))
```

Output:

```
[example.py:8] requesting user data  
[example.py:13] got bad response, 'tes '  
[example.py:18] user, Donald, is playing
```

From the log can you tell what happened when our code was executed?

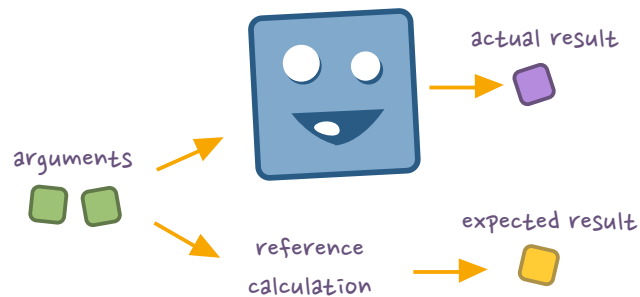
Good logging is an art but there are some best practices we can follow that can help us improve our logs and make them more useful:

1. Logs should contain everything a programmer needs to debug a problem. You shouldn't just report an error code - instead you should also include the conditions that caused the error.
2. Logs shouldn't contain so much information that reading the logs becomes impossible.
3. Information concerning a single event should not be split on separate lines or entries.
4. Errors reported to the user should also be logged.
5. Logs should use categories to differentiate different kinds of activities.

## 10.4 Unit Testing

Testing and debugging approaches we have mentioned thus far consider the internal consistency and state of the program. Another useful way of thinking about software is as a black box where for a specific input, we expect a specific result.

External tests don't rely on any understanding of program state or how a function works. Instead, we independently calculated an expected result for a function given a specific set of arguments and compare that to the actual result returned by the function:



We can instrument these expectations as tests that we run on the code. When the code we are testing is relatively small (like a single function call or just a few lines of code that accomplish a distinct task) we call the result a **unit test**.

While there are several excellent testing frameworks for Python, we will demonstrate one called `pytest` which is particularly friendly and versatile. Instructions for installing `pytest` were provided in [Chapter 2](#).

Our first test is a bit silly - we have a function that intends to add two to a value but it has a bug in it. The test is ignorant of how the function works - it simply compares the expected result with the actual result:

```

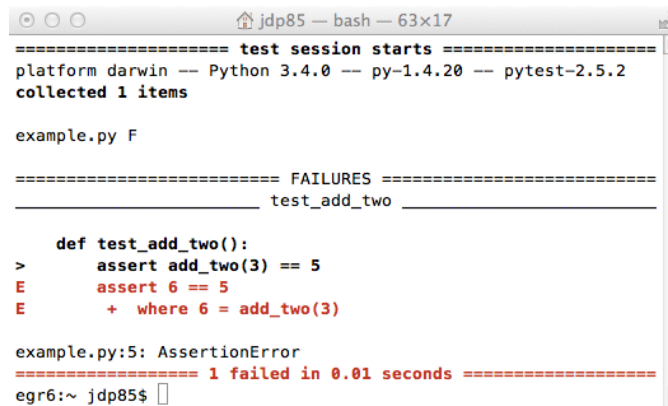
1 def add_two(x):
2     return x + 3
3
4 def test_add_two():
5     assert add_two(3) == 5

```

Note that we did not include the `pytest` module in our code! One feature of `pytest` is we can write tests that are ignorant of any special `pytest` classes or functionality. We can then launch `pytest`, which will search our code for unit tests to run by typing this at the command line:

```
$ py.test test*.py
```

`pytest` will then look for functions with the prefix “`test_`” and run them. If a test fails then it notifies the developer with a unit test report that looks like this:



```

jdp85 ~ — bash — 63x17
===== test session starts =====
platform darwin -- Python 3.4.0 -- py-1.4.20 -- pytest-2.5.2
collected 1 items

example.py F

===== FAILURES =====
_____ test_add_two _____

    def test_add_two():
>     assert add_two(3) == 5
E       assert 6 == 5
E       + where 6 = add_two(3)

example.py:5: AssertionError
===== 1 failed in 0.01 seconds =====
egr6:~ jdp85$

```

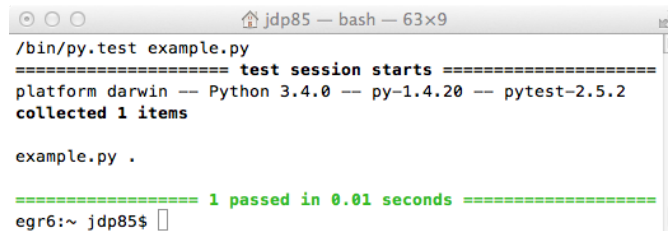
In this case, our `add_two()` function was flawed. We can fix it easily enough:

```

1 def add_two(x):
2     return x + 2

```

And then run `pytest` again to get this report:



```

jdp85 ~ — bash — 63x9
/bin/py.test example.py
===== test session starts =====
platform darwin -- Python 3.4.0 -- py-1.4.20 -- pytest-2.5.2
collected 1 items

example.py .

===== 1 passed in 0.01 seconds =====
egr6:~ jdp85$

```

While you could easily run tests “by hand”, testing frameworks provide consistency, convenience, and detailed readable information about how collections of tests fail that running ad hoc tests don’t. The automation gained in a testing framework becomes key to doing regression testing.

## 10.5 Equivalence Classes

In most cases it’s not practical or even possible to exhaustively test the complete domain of values passed to a function nor is it possible to test the complete range of values returned from a function.

When we can’t exhaustively test a function, we still can identify tests that give us confidence about the behavior of a function. One common approach creates a partitioning of the function’s problem space where we expect the behavior of a function to be similar.

We can define these partitions relative to the problem by finding properties in the problem's domain or range where we can observe similar behavior.

We can also define these partitions relative to working code by examining similarities in the execution path through the code toward a result. How well tests exercise the different paths through a program is called code coverage.

Consider this code that determines if a value is even or odd:

```
1 def is_odd(x):  
2     return x % 2 == 1
```

The equivalence classes we might consider for this code include:

- even positive numbers
- odd positive numbers
- even negative numbers
- odd negative numbers
- zero

Once we have devised appropriate equivalence classes, we select one or more represent members from each class. The number of samples we choose from each class tends to be a compromise between what's practical and the confidence we need to establish.

```
1 def test_even_positive(x):  
2     assert is_odd(8) == False  
3     assert is_odd(1024) == False  
4  
5 def test_odd_positive(x):  
6     assert is_odd(7) == True  
7     assert is_odd
```

## 10.6 Regression Testing

One problem that developers often face is that by adding or modifying code we might actually introduce new bugs - even in code that at one point worked! When a new feature or update causes existing functionality to be defective we call it a **regression**.

Since unit tests are designed to be repeatable, we can use them as a tool to detect regressions. If we design a number of tests that succeed with our current

code, if we add new or modified code we can run the old tests to provide some assurance that the new code doesn't change or modify the behavior of our software in undesirable ways. This process of running old tests after we make changes to the software is called **regression testing**.

Imagine that we added a new function called `add_four()` and a misguided software developer implemented it like this:

```
1 def add_two(x):
2     return x + 4
3
4 def add_four(x)
5     return add_two(x)
```

If we created unit tests for `add_four()` they would pass! But we have a regression: `add_two()` doesn't work any more. By running the tests we already developed for `add_two()`, we can detect this regression.

**Principle 9** (Test Early and Often). *Testing is a mechanism to build confidence in the correctness of our solution. Testing early and often encourages robust software development throughout the development process.*

Regression testing is a simple but powerful idea. By building good tests, you are creating a measure of confidence that your code works correctly both in the present and in the future.

## 10.7 Test Driven Development

While testing is often done after there is code to test, test driven development suggests that we write tests for code before we write the code itself!

This might seem odd at first but it's a very powerful way of thinking about software. Instead of thinking about how we solve the problem first, we think about what the solution to a particular problem should look like. We then solve the problem "by hand" to create tests. As we create tests we think antagonistically - what tests can we create that would convince us that the solution is probably correct?

Once the tests have been designed, we implement behavior and incrementally test the behavior. When the solution passes every test, we are done with the implementation. The major insight that test driven development offers is that by focusing on testing we might build more reliable software.

**Principle 10** (Drive Development with Tests). *By designing tests early, we force ourselves to understand the problem and when we write code we do so to accomplish concrete goals.*

If we wanted to write a function to compute the numbers in the fibonacci sequence, we might begin by stubbing out the `fibonacci()` function and then writing our tests:

```
1 def fibonacci(n):
2     "Return the nth fibonacci"
3     return 1
4
5 def test_fibonacci(x)
6     assert fibonacci(0) == 1
7     assert fibonacci(1) == 1
8     assert fibonacci(2) == 2
9     assert fibonacci(10) == 89
10    # and so on..
```

If you run `pytest` on this sample you'll find one test passes and the rest fails. Once we have sufficient test coverage, then we begin implementing the fibonacci function such that it passes our tests. This fundamentally changes the way that we frame the problem and encourages developers to understand the problem better before devising a solution. It also sets a more concrete goal for completing a task - a function is "complete" when it can pass all of the tests.

## Glossary

**bug** See **software defect**.

**contractual design** A design methodology where we think about the relationship between objects, methods and functions as being contracts with certain requirements and assurances.

**logging** The practice of communicating information about the state and progress of a program as textual messages.

**exception** A kind of error that can be specially handled within Python.

**invariant** A kind of contract that must always be True during a process.

**postcondition** A kind of contract that must be True once a process has been executed.

**precondition** A kind of contract that must be True before a process is executed.

**regression testing** A testing procedure where old tests are run after new code has been added to help ensure existing functionality has not been changed.

**software defect** A mistake or error in code that causes incorrect or undefined behavior.

**test driven development** A development philosophy where tests are designed before the code they are intended to test.

**unit testing** A test that checks a small unit of code for correct behavior.

### Reading Questions

- 10.1. Why is testing important?
- 10.2. What is the fail early principle?
- 10.3. What is the risk of trying to “fix” the state of the system rather than failing when inconsistent state is recognized?
- 10.4. What’s the relationship between a software defect and inconsistent state?
- 10.5. What three contracts types are features of contractual design?
- 10.6. How is contractual design related to the fail early principle?
- 10.7. Is contractual design a first class part of the Python language?
- 10.8. What is an exception?
- 10.9. What is a unit test?
- 10.10. How does pytest help us test software?
- 10.11. How can pytest be used as part of regression testing?
- 10.12. Besides testing our software, what secondary benefits come with unit testing?
- 10.13. What steps do we take in test driven development?
- 10.14. How can logging help us test and debug software?
- 10.15. Why is logging better than using print statements to convey information about how a program is being evaluated?
- 10.16. Name five different logging levels.
- 10.17. Should testing only be done after we finish writing software?



- 10.18. What approach does Test Driven Development apply to implementing tests?
- 10.19. What is a benefit of using exception handling?
- 10.20. What is a bug or software defect?

### Exercises

#### 10-1. Zodiac sign II

Revisit Problem 5-7 with contractual programming in mind. Design and implement preconditions and postconditions for the problem.

#### 10-2. Time since midnight III

Revisit Problem 5-10 with contractual programming in mind. Design and implement precondition and postconditions for the problem.

#### 10-3. Temperature conversion II

Revisit Problem 5-18 with contractual programming in mind. Design and implement precondition and postconditions for the problem.

#### 10-4. SMS Language expander III

Design six unit tests using pytest for the SMS language expander in Problem 6-2.

#### 10-5. Winning at tic-tac-toe II

Design four unit tests using pytest for the tic-tac-toe function you wrote in Problem 6-7.

#### 10-6. Parsing time III

Describe equivalence classes for Problem 6-9 (or Problem 9-9). Using these equivalence classes design test instances you can implement with pytest.

#### 10-7. Parsing calendar dates III

Describe equivalence classes for Problem 6-10 (or Problem 9-10). Using these equivalence classes design test instances you can implement with pytest.

#### 10-8. Lights out II

Describe equivalence classes for Problem 6-11. Using these equivalence classes design test instances you can implement with pytest.

#### 10-9. Matrix multiplication III

Describe equivalence classes for Problem 6-12. Using these equivalence classes design test instances you can implement with pytest.

**10-10. Matrix multiplication IV**

Problem **9-11** had several methods associated with it. Think about equivalence classes for each of these methods and design test instances you can implement with `pytest`.

**10-11. Match 3 II**

Problem **9-15** is a relatively complex game. Add logging to your solution such that we can trace events as they happen. You should be able to read the resulting log and understand what is happening within your program.

**10-12. Parsing integers**

Write a function called `parse_int()` that parses an integer and returns the resulting `int` value without using Python's `int()`! Use test-driven design to start with test cases and then work toward a solution that satisfies these test cases (consult the railroad diagram for `int` literals in Section **3.3**). Hint: You'll need to look at each digit one at a time and use an accumulation pattern to build up the number to be returned.

**Chapter Notes**

Function preconditions and postconditions were first proposed by Tony Hoare in 1969. Design by Contract (TM) was first coined by Bertrand Meyer in 1986 who promoted a paradigm of software interaction modeled after business contracts. Meyer developed this approach in the Eiffel programming language which has first class support for software contracts.

There are several excellent unit testing frameworks for Python including `pytest`, `nose`, and `pyunit`. While `pyunit` is bundled with the standard Python distribution we cover `pytest` in this chapter because its very easy to explain and in the simplest kinds of tests you can be ignorant of the `pytest` API.

[63] Peter Wayner. Smithsonian honors the original bug in the system. *New York Times Cybertimes Extra*, dec 1997. [link](#).

# 11

## Regular Expressions \*

*Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems.*

*Jamie Zawinski*

```
1 overview = ["Concatenations",
2             "Alternations",
3             "Repetition",
4             "Groups",
5             "Splitting_Strings",
6             "Search_and_Replace",
7             "Glossary",
8             "Reading_Questions",
9             "Exercises"
10            "Chapter_Notes"]
```

A regular expression is a sequence of characters used to match a particular patterns of characters. Regular expressions are really their own mini-language which define rules for matching text.

In previous chapters we have written programs to check if a phone-number, email address, or date is in the correct format. Regular expressions can do these kinds of tasks much more succinctly and often times more quickly! Regular expressions are optimized and compiled into a byte-code which is evaluated by a matching engine written in C.

In Chapter X we discussed how Turing's results on computability meant that if something was computable then computers could compute them. Specifically, Turing's result concerned language. A language that can compute anything that is computable is called Turing complete. But many languages, like regular expressions, aren't Turing complete. The consequence as far as we are concerned for the moment is that regular expressions can't match every possible pattern.

Instead, regular expressions can match three kinds of patterns:

1. concatenations, where one string may be matched after another,
2. alternations, where one string may be matched instead of another, and
3. repetitions<sup>1</sup>, where one string may be matched multiple times.

## 11.1 Concatenations

The simplest sort of regular expression is a sequence of letters or numbers that should be matched. Consider this example where we are looking for a match to the word 'blatant'.

```

1 import re
2
3 re.match('blatant', 'blatantly') → <Match object>
4 re.match('blatant', 'blaze') → None

```

'blatant' is a prefix of 'blatantly' so it matches but it doesn't match 'blaze'. We call this a concatenation because each letter is matched one after another.

We can also specify the character to match as part of a character class defined within square brackets<sup>2</sup>. Character classes are formed from individual characters or ranges of characters. The character class to match any uppercase letter is [A-Z]. The character class to match a letter of any case is [A-Za-z]. The character class to match a numeric digit is [0-9]. The character class to match just vowels is [aeiou].

Lets use character classes to match two spellings of the word gray:

```

1 import re
2
3 re.match('gr[ae]y', 'gray') → <Match object>

```

1. The theoretical name for this is Kleene star.

2. Technically character classes are a form of alternation.

```
4 re.match('gr[ae]y', 'grey') → <Match object>
```

Certain character patterns are very common have escape sequence short cuts. Some of the most common are listed in Table 111.

Table 111: *Common Regular Expression Sequences*

Sequence	Description
<code>\d</code>	Match a decimal digit character [0-9]
<code>\D</code>	Match any non-digit character [^0-9]
<code>\s</code>	Match whitespace character [ \t\n\r\f\v]
<code>\S</code>	Match any non-whitespace character [^\t\n\r\f\v]
<code>\w</code>	Match any alpha-numeric character [a-zA-Z0-9_]
<code>\W</code>	Match any non-alpha-numeric character [a-zA-Z0-9_]

One complication of these sequences is that they don't live in the same world of escape sequences we already know for strings. Normal strings need to escape the sequences for regular expressions:

```
1 re.match('apple\sbanana', 'apple_banana') →
  <Match object>
```

Since this could quickly get confusing, Python provides a string literal just for regular expressions which starts with an 'r' prefix. Rewriting the last example we get:

```
1 re.match(r'apple\sbanana', 'apple_banana') →
  <Match object>
```

## 11.2 Alternations

Alternation is accomplished with the '|' symbol. You can read an alternation as an 'or' since it matches one thing or another.

```
1 re.match(r'red|gray', 'red') → <Match object>
2 re.match(r'red|gray', 'gray') → <Match object>
```

Parenthesis may be used to group an alternation:

```
1 re.match(r'un(ru|weild)ly', 'unruly') → <Match object>
```

### 11.3 Repetition

There are four constructs for repetition summarized in Table 112.

Table 112: *Regular Expression Repetition*

Sequence	Description
*	Match 0 or more times
+	Match 1 or more times
?	Match 0 or 1 times
{m, n}	Match at least m times and at most n times

These patterns control how many times a sequence may appear in a match:

```
1 re.match(r'\d+', '5') → <Match object>
2 re.match(r'\d+', '5231') → <Match object>
3 re.match(r'\d{3}-\d{3}-\d{4}',
4         '123-456-7890') → <Match object>
```

And like alternations, parenthesis can be used to group repeating patterns:

```
1 re.match(r'ba(na){2,}', 'banana') → <Match object>
2 re.match(r'ba(na){2,}', 'banananana') → <Match object>
3 re.match(r'analog(ue)?', 'analogue') → <Match object>
```

### 11.4

#### Exercises

##### 11-1. Encheferization

Write a program that converts English to Chef Speak. The user should input a sentence and your program should return the Chef Speak equivalent by following these rules:

- a) Make these substring replacements if the match appears at the beginning of a word:
  - i.an → un
  - ii.au → oo
  - iii.a → e

iv.en → e  
v.e → i  
vi.the → zee  
vii.th → t  
viii.v → f  
ix.w → v

- b) Make these substring replacements as long as the match does not appear at the beginning of a word:

i.ew → oo  
ii.e → e-a  
iii.f → ff  
iv.ir → ur  
v.i → ee  
vi.ow → oo  
vii.o → u  
viii.tion → shun  
ix.u → oo

- c) Each sentence should end with, "Bork! Bork!"





A

# Pair Programming

Coming soon. Until then read [All I Really Need to Know about Pair Programming I Learned In Kindergarten](#).



**B**

# Turtle Graphics

Coming soon. Until then read [Python Turtle Documentation](#)



## C

# Unicode

Unicode is a standard that maps characters used in natural language and human communication into numeric values (called code points). The Unicode standard supports most modern scripts and many extinct ones. At the time this text was written over 250,000 code points have been assigned in Unicode with roughly 860,000 code points unassigned.

We have listed the first 128 code points (which also correspond to the first 128 characters in [ASCII](#)) for reference<sup>1</sup>.

Table C1: *Basic Latin*

Dec	Python	Unicode Name
0	<code>\u0000</code>	Null Character (NUL)
1	<code>\u0001</code>	Start of Header (SOH)
2	<code>\u0002</code>	Start of Text (SOT)
3	<code>\u0003</code>	End of Text (ETX)
4	<code>\u0004</code>	End of Transmission (EOT)
5	<code>\u0005</code>	Enquiry (ENQ)
6	<code>\u0006</code>	Acknowledgment (ACK)
7	<code>\a</code>	Bell (BEL)
8	<code>\b</code>	Backspace (BS)
9	<code>\t</code>	Horizontal tab (HT)
10	<code>\n</code>	Line Feed (LF)
11	<code>\v</code>	Vertical tab (VT)

---

1. The full database of Unicode code points is listed at <http://unicode.org/charts/>. A nicely done interactive table of these values is maintained at <http://unicode-table.com/en/>.

Dec	Python	Unicode Name
12	\f	Form Feed (FF)
13	\r	Carriage Return (CR)
14	\u000e	Shift Out (SO)
15	\u000f	Shift In (SI)
16	\u0010	Data Link Escape (DLE)
17	\u0011	Device Control 1 (DC1)
18	\u0012	Device Control 2 (DC2)
19	\u0013	Device Control 3 (DC3)
20	\u0014	Device Control 4 (DC4)
21	\u0015	Negative Acknowledgment (NAK)
22	\u0016	Synchronous idle (SYN)
23	\u0017	End of Transmission Block (ETB)
24	\u0018	Cancel (CAN)
25	\u0019	End of Medium (EM)
26	\u001a	Substitute (SUB)
27	\u001b	Escape (ESC)
28	\u001c	File Separator (FS)
29	\u001d	Group Separator (GS)
30	\u001e	Record Separator (RS)
31	\u001f	Unit Separator (US)
32	space	Space
33	!	Exclamation Mark
34	"	Quotation Mark
35	#	Number Sign
36	\$	Dollar Sign
37	%	Percent Sign
38	&	Ampersand
39	'	Apostrophe
40	(	Left Parenthesis
41	)	Right Parenthesis
42	*	Asterisk
43	+	Plus Sign
44	,	Comma
45	-	Hyphen-Minus

Dec	Python	Unicode Name
46	.	Full Stop
47	/	Solidus
48	0	Digit Zero
49	1	Digit One
50	2	Digit Two
51	3	Digit Three
52	4	Digit Four
53	5	Digit Five
54	6	Digit Six
55	7	Digit Seven
56	8	Digit Eight
57	9	Digit Nine
58	:	Colon
59	;	Semicolon
60	<	Less-Than Sign
61	=	Equals Sign
62	>	Greater-Than Sign
63	?	Question Mark
64	@	Commercial At
65	A	Latin Capital Letter A
66	B	Latin Capital Letter B
67	C	Latin Capital Letter C
68	D	Latin Capital Letter D
69	E	Latin Capital Letter E
70	F	Latin Capital Letter F
71	G	Latin Capital Letter G
72	H	Latin Capital Letter H
73	I	Latin Capital Letter I
74	J	Latin Capital Letter J
75	K	Latin Capital Letter K
76	L	Latin Capital Letter L
77	M	Latin Capital Letter M
78	N	Latin Capital Letter N
79	O	Latin Capital Letter O

Dec	Python	Unicode Name
80	P	Latin Capital Letter P
81	Q	Latin Capital Letter Q
82	R	Latin Capital Letter R
83	S	Latin Capital Letter S
84	T	Latin Capital Letter T
85	U	Latin Capital Letter U
86	V	Latin Capital Letter V
87	W	Latin Capital Letter W
88	X	Latin Capital Letter X
89	Y	Latin Capital Letter Y
90	Z	Latin Capital Letter Z
91	[	Left Square Bracket
92	\\	Reverse Solidus
93	]	Right Square Bracket
94	^	Circumflex Accent
95	_	Low Line
96	`	Grave Accent
97	a	Latin Small Letter A
98	b	Latin Small Letter B
99	c	Latin Small Letter C
100	d	Latin Small Letter D
101	e	Latin Small Letter E
102	f	Latin Small Letter F
103	g	Latin Small Letter G
104	h	Latin Small Letter H
105	i	Latin Small Letter I
106	j	Latin Small Letter J
107	k	Latin Small Letter K
108	l	Latin Small Letter L
109	m	Latin Small Letter M
110	n	Latin Small Letter N
111	o	Latin Small Letter O
112	p	Latin Small Letter P
113	q	Latin Small Letter Q



Dec	Python	Unicode Name
114	r	Latin Small Letter R
115	s	Latin Small Letter S
116	t	Latin Small Letter T
117	u	Latin Small Letter U
118	v	Latin Small Letter V
119	w	Latin Small Letter W
120	x	Latin Small Letter X
121	y	Latin Small Letter Y
122	z	Latin Small Letter Z
123	{	Left Curly Bracket
124		Vertical Line
125	}	Right Curly Bracket
126	~	Tilde
127	\u007f	Delete



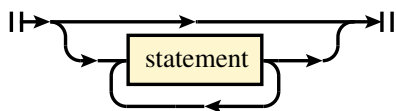
## D

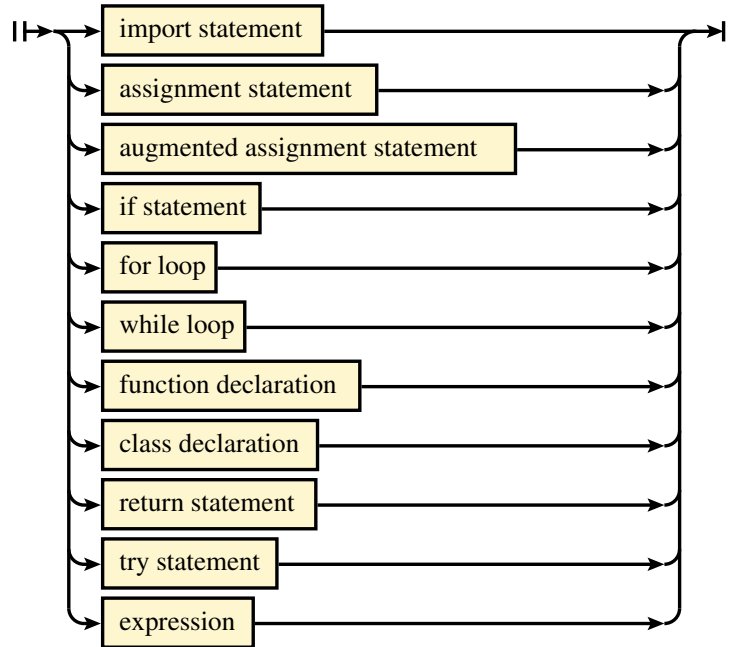
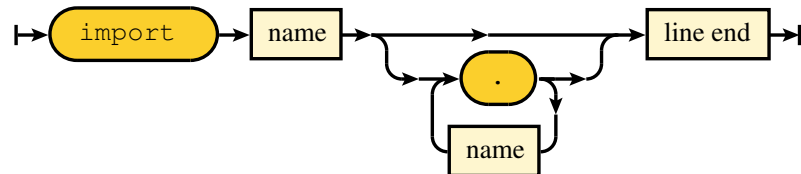
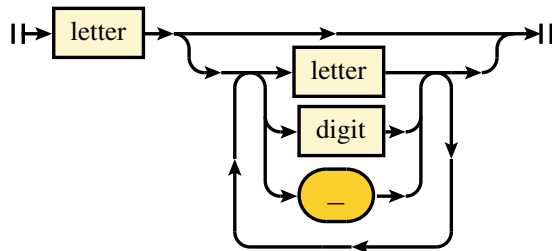
# Syntax Diagrams

In this text we have presented a simplified subset of the Python language using syntax diagrams (also called railroad diagrams). These diagrams define a syntax similar to the one Python uses internally to parse Python programs into lower level instructions.

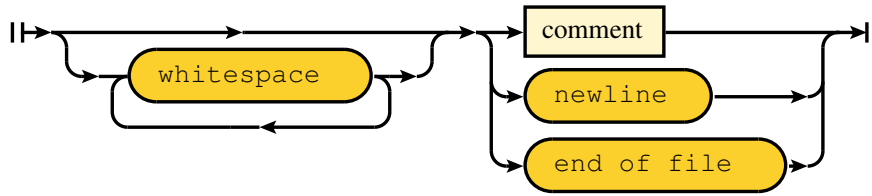
We encourage the reader to compare the syntax diagrams presented here to the actual grammar used by the Python parser. Note that the [Python grammar](#) is given in a formal description language derived from [Extended Backus–Naur Form](#) (EBNF).

**program**

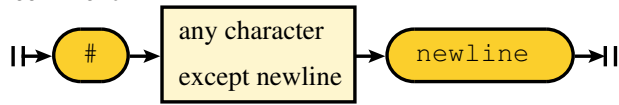


**statement****import statement****name**

**line end**



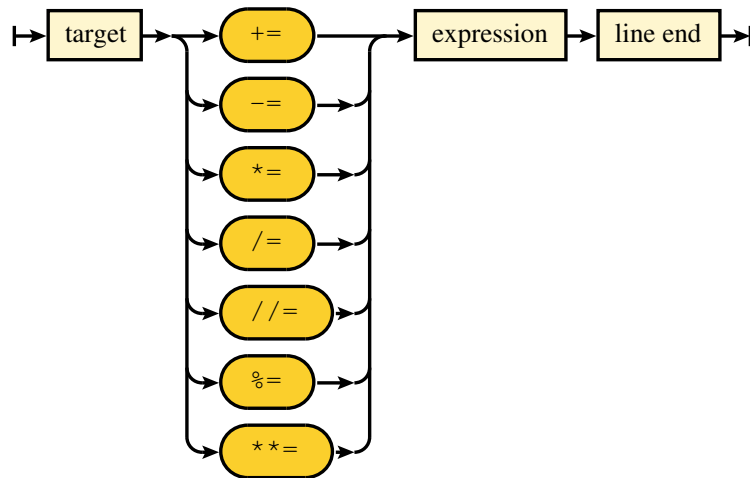
**comment**

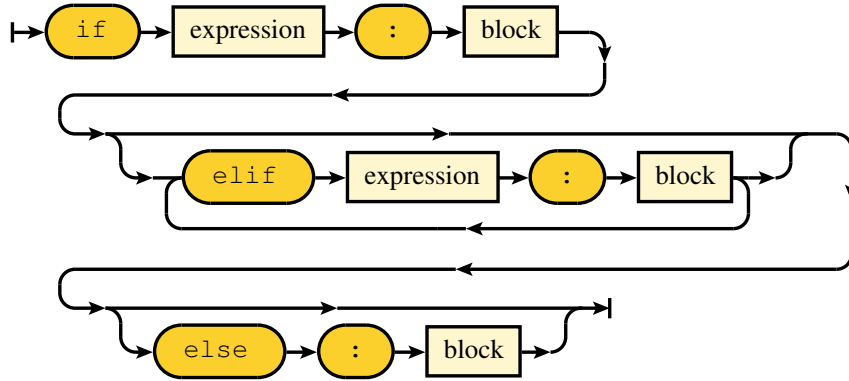
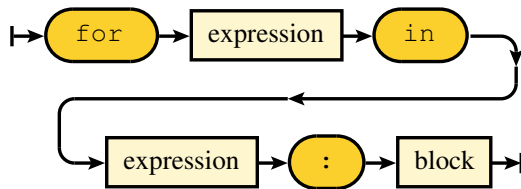
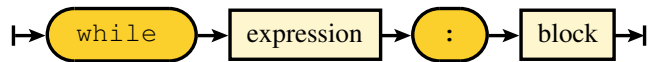
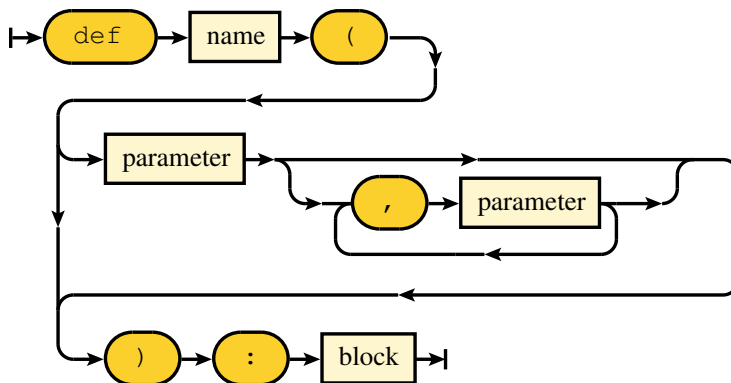


**assignment statement**

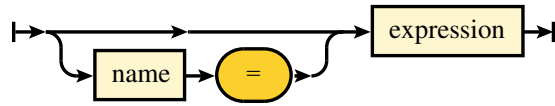


**augmented assignment statement**

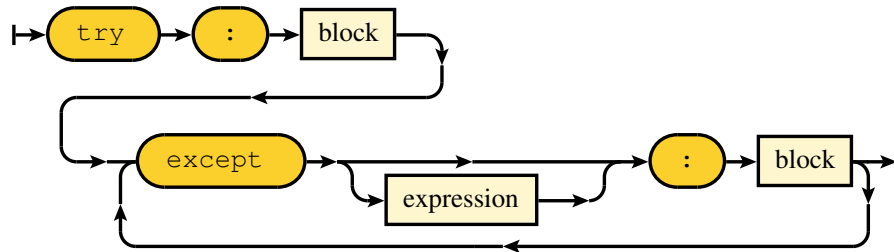


**if statement****for loop****while loop****function declaration**

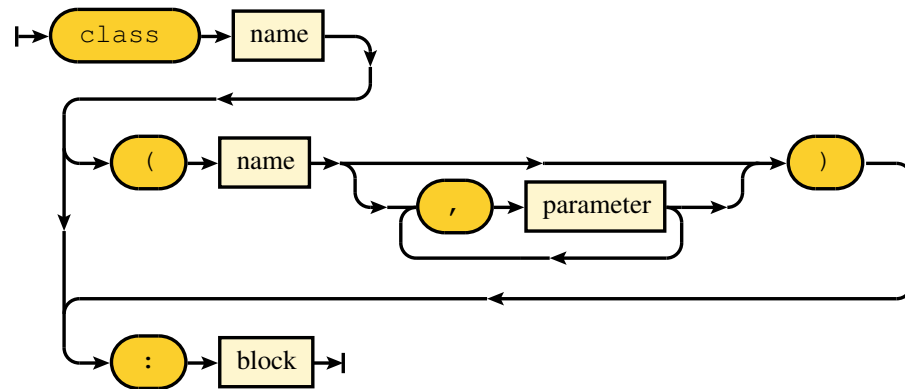
**parameter**



**try statement**

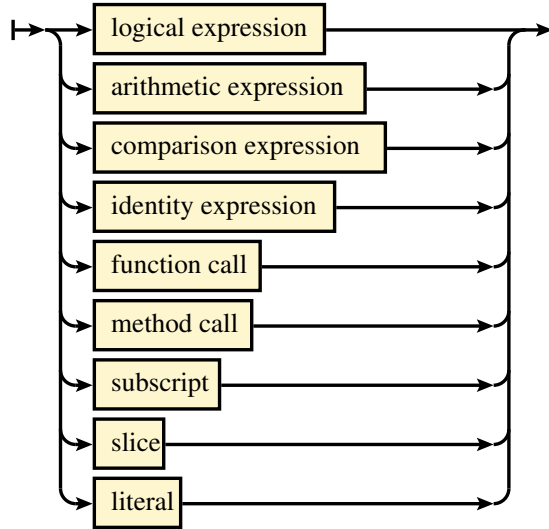
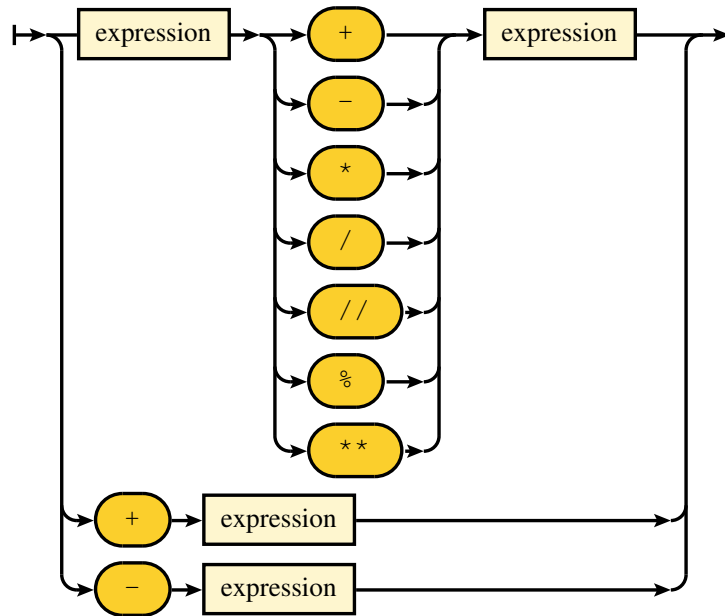


**class declaration**



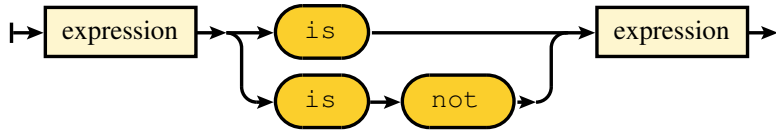
**return statement**



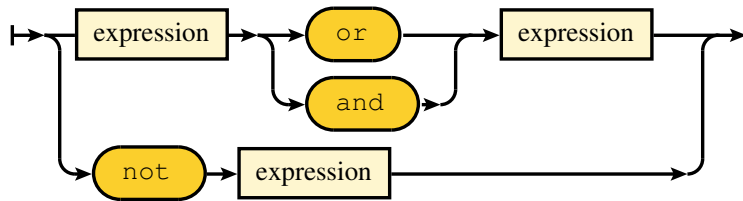
**expression****arithmetic expression**



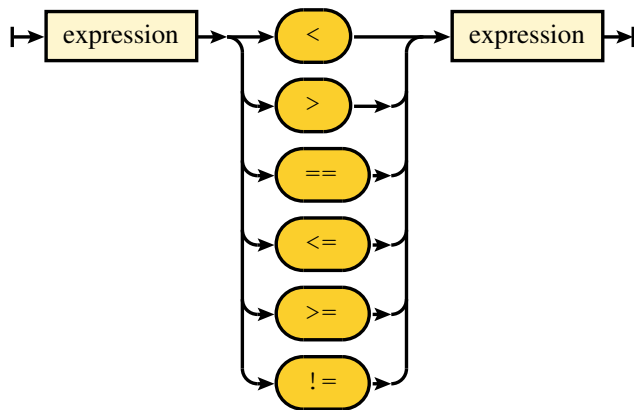
**identity expression**



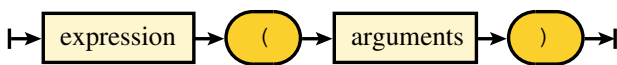
**logical expression**



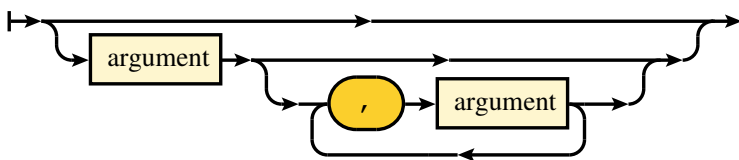
**comparison expression**

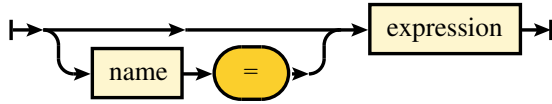
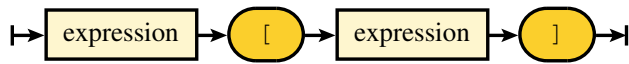
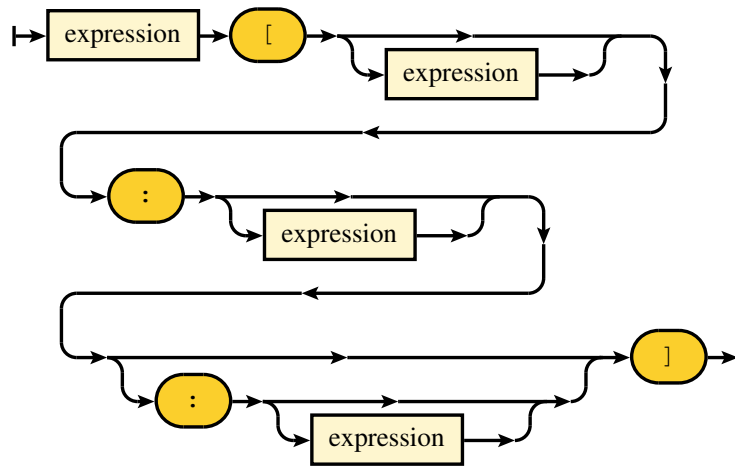
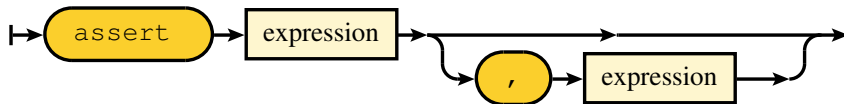


**function call**

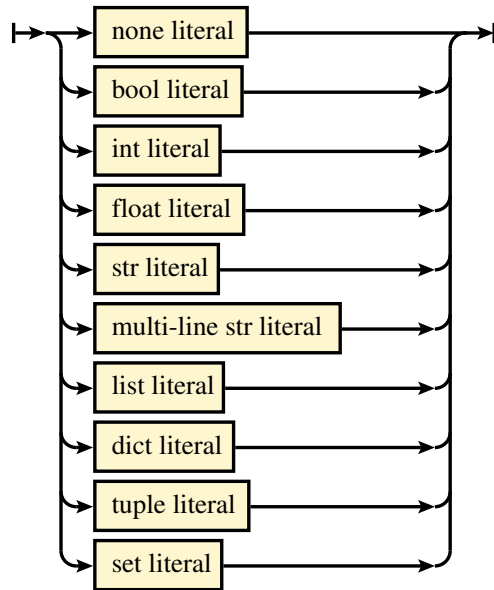


**arguments**

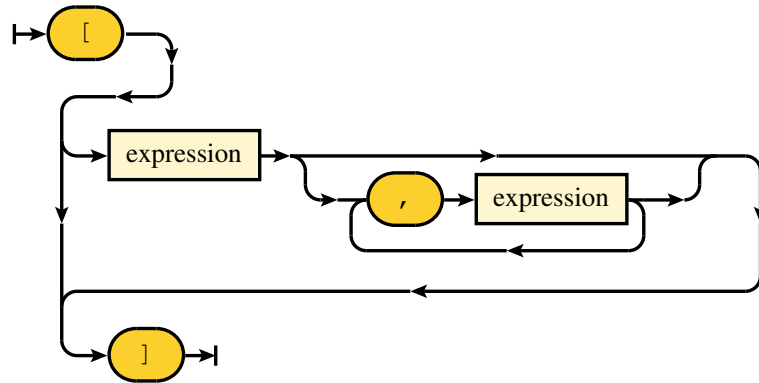


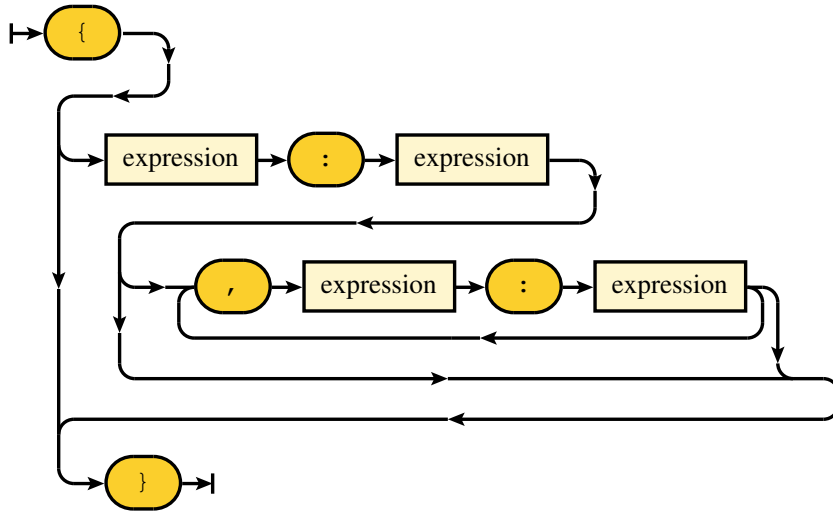
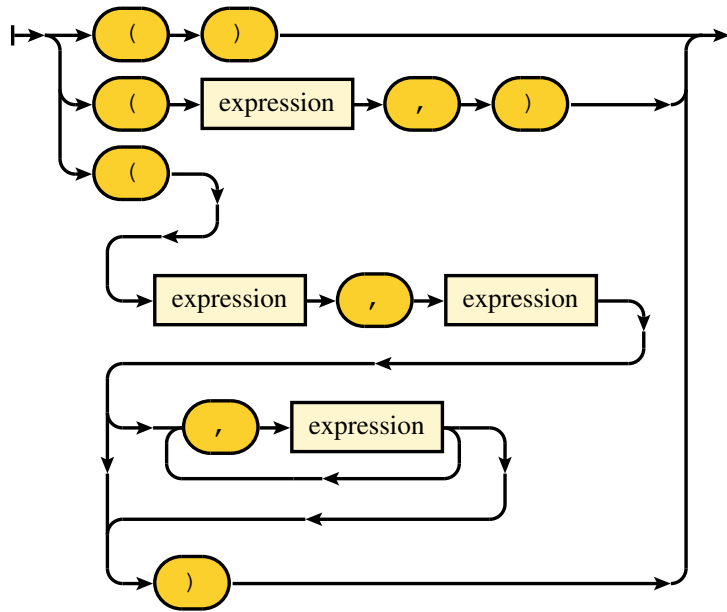
**argument****method call****subscript****slice****assertion**

**literal**

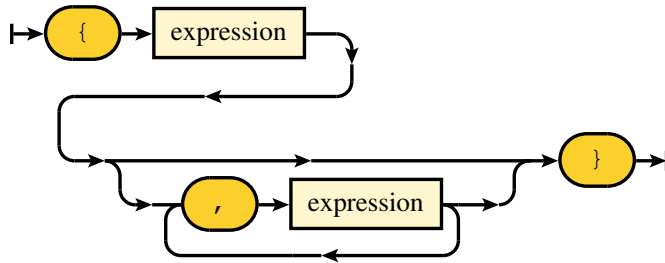


**list literal**

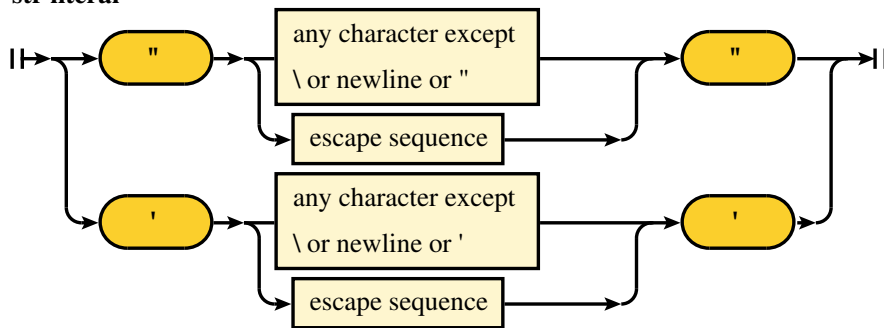


**dict literal****tuple literal**

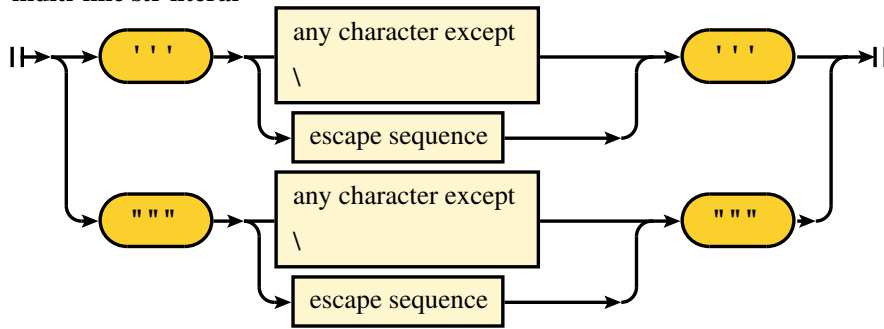
**set literal**



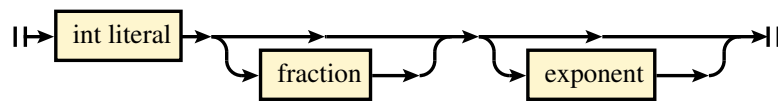
**str literal**



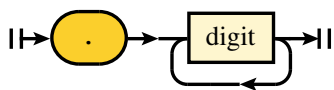
**multi-line str literal**

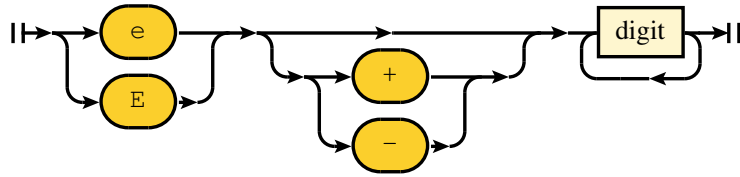
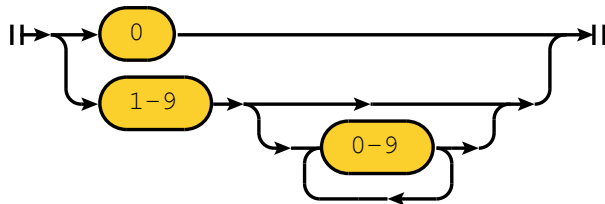
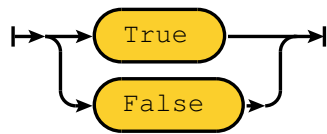


**float literal**



**fraction**



**exponent****int literal****bool literal****none literal**

# List of Strategies

1	Step (Understand the Problem)	10
2	Step (Plan a Solution)	11
3	Step (Implement and Test)	13
4	Step (Reflection)	15
1	Strategy (Break Problems Into Cases)	58
2	Strategy (Identify and Factor Out Repetition with Functions)	120
3	Strategy (Accumulation Pattern)	162

# List of Principles

1	Principle (Comments Should Explain Why)	40
2	Principle (Code Should Read Like Poetry)	54
3	Principle (Functions Should Do One Thing)	116
4	Principle (Avoid Global Variables)	203
5	Principle (Fail Early)	213
6	Principle (Fail On Your Own Terms)	214
7	Principle (Think Contractually)	215
8	Principle (Use Logs to Tell a Story)	218
9	Principle (Test Early and Often)	224
10	Principle (Drive Development with Tests)	225



# List of Tables

31	Logical Operators . . . . .	56
32	Logical Operator Truth Table . . . . .	56
33	Arithmetic Operators . . . . .	60
34	Comparison Operators . . . . .	63
35	Arithmetic Functions . . . . .	65
36	Literal String Syntax . . . . .	68
37	String Escape Sequences . . . . .	69
38	String Operators . . . . .	71
39	String Methods . . . . .	73
310	Conversion Functions . . . . .	76
41	Operator Precedence . . . . .	92
42	Operator Precedence Examples . . . . .	92
43	A Few Common Format Codes . . . . .	99
61	Common List Operations . . . . .	141
62	Common Dictionary Operations . . . . .	147
63	Common Tuple Operations . . . . .	148
64	Common Set Operations . . . . .	151
71	Range Parameters . . . . .	165
72	Newton's Method . . . . .	169
81	Common File Access Modes . . . . .	180
82	Common File Methods . . . . .	180
91	Some Common Exceptions . . . . .	201
101	Log Categories . . . . .	218

111	Common Regular Expression Sequences . . . . .	231
112	Regular Expression Repetition . . . . .	232
C1	Basic Latin . . . . .	239

# Index

## A

Abelson, Hal, 9  
accuracy, 64  
argument, 110  
assertion, 214  
assignment, 51

## B

Babbage, Charles, 3  
batteries included, 31  
behavior, 193  
block, 57  
bool, 55  
bug, 212

## C

class, 192  
clock arithmetic, 61  
closure, 125  
code block, 57  
comments, 14, 39  
comparison operations, 62  
compiler, 6  
computer, 2  
contract, 214  
contractual design, 214

## D

data type, 49

dedented, 57  
defect, 212  
dictionary, 145  
dining philosophers problem, 24  
docstring, 115  
dynamic typing, 52

## E

Easier to Ask Forgiveness than Per-  
mission (EAFP), 202  
eight queens problem, 23  
encapsulation, 195  
ENIAC, 5  
escape sequences, 69

## F

False, 55  
field, 194  
Fields, 194  
first-class function, 124  
float, 59  
floating point, 59  
for loop, 161  
function, 109

## H

Hollerith, Herman, 5  
Hopper, Grace, 6

**I**

identifier, 52  
IDLE, 36  
if statement, 56  
import, 65  
inconsistent state, 213  
indented, 57  
infinite loop, 169  
infinite recursion, 124  
infix expression, 51  
inheritance, 197  
input function, 38  
int, 59  
integer division, 61  
invariant, 215  
iteration, 161

**K**

keyword, 53

**L**

list, 139  
list of lists, 143  
literal, 51  
logical operators, 55  
Look Before You Leap (LBYL), 202  
Lovelace, Lady August Ada, 4

**M**

machine independent programming, 6  
method, 73, 193  
Methods, 193  
missionaries and cannibals problem, 22  
module, 65  
Moore, Gordon, 5  
Moore's Law, 5

**N**

nested lists, 143  
None, 75  
numbers, 58

**O**

object, 192  
object-oriented programming, 31

**P**

parameter, 110  
Pascal, Blaise, 2  
Pascaline calculator, 2  
PATH, 34  
Pólya, George, 9  
polymorphism, 199  
postcondition, 215  
postfix expression, 51  
precision, 64  
precondition, 215  
prefix expression, 51  
print function, 38  
private, 195  
program, 38  
public, 195  
Python, 30  
Python Enhancement Proposal (PEP), 45  
Python, installation, 32

**R**

Read the fine manual (RTFM), 45  
recursion, 123  
refactoring, 16  
reflection, 116  
Rossum, Guido van, 30

**S**

scope, 116

short-circuit, 93  
slicing, 71  
software development process, 9  
state, 194  
static typing, 52  
str, 67  
strings, 67  
subclass, 197  
subscript operator, 70  
subscripting, 71  
substring, 70

**T**

Therac-25, 212  
top-down design, 121  
Tower of Hanoi problem, 23  
traveling salesman problem, 24  
Trémaux, Charles, 16  
True, 55  
truth table, 56  
truthiness, 95  
tuple, 147  
tuple unpacking, 149  
Turing Machine, 6  
Turing, Alan, 6  
two generals' problem, 24  
`type()` function, 77

**U**

Unified Modeling Language (UML),  
192

**V**

value, 50  
variable, 50  
variable naming, 52

**W**

while loop, 168



# Authors & Contributors

James Dean Palmer (<http://jdpalmer.org>) is an associate professor in the Computer Science program at Northern Arizona University. Dr. Palmer received a B.S. in Computer Engineering at Texas A&M, an M.S. in Computer Visualization at Texas A&M, and a Ph.D. in Computer Science at the University of Texas Dallas. His research interests include computer languages, visualization, geometry, and computer science pedagogy.

## Contributors

Jason Hedlund developed many of the reading questions, exercise sample runs, exercise sample solutions, and typeset many of the labs and the slides.

## Bug fixes

I would like to thank everyone that has submitted bug fixes, pointed out errors in the code or providing other valuable feedback that has made the text what it is!

Jeffrey Covington

Steve Jacobs





# Colophon

The image on the cover is the silhouette of a human head with a directed graph masked within. The logo font begins with Nanum Pen Script and ends with Century Gothic. The book title, head logo, and title design were created by James Palmer and should not be used without permission.

Problemspace was composed with [L<sup>A</sup>T<sub>E</sub>X](#) and [E<sup>T</sup><sub>E</sub>X](#) using a modified version of the [L<sup>A</sup>T<sub>E</sub>X](#)Book layout produced by the [The Editorium](#). Hand drawn figures were produced with [Sketch](#). Railroad diagrams were produced by D. Richard Hipp's [bubble-generator](#). Expression trees were produced with [Graphviz](#).