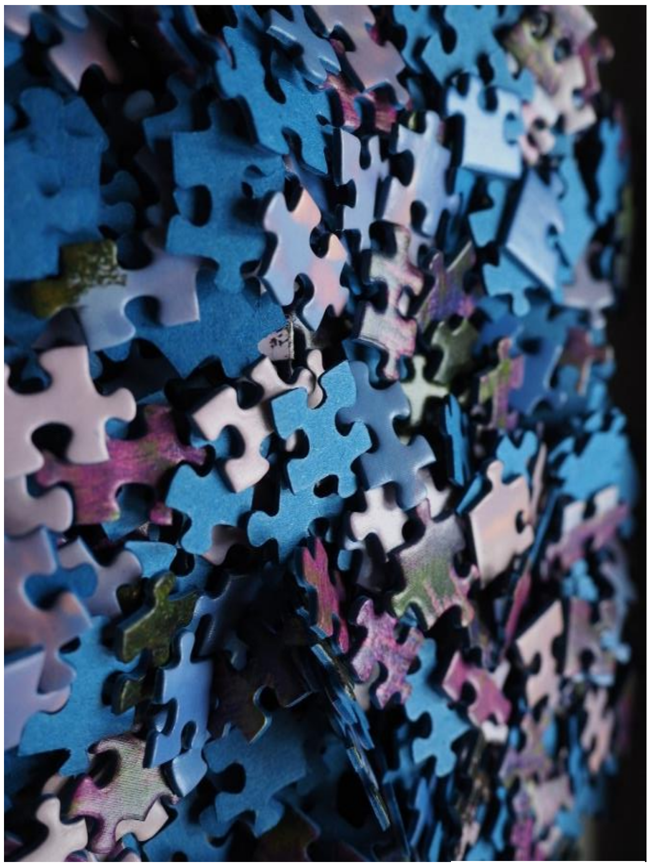

www.maxpixel.net

PROGRAMMING FOR PROBLEM SOLVING

Version 1.0 13 July 2018

ABSTRACT

This book provides an introduction to computer programming using Python as a way to solve problems. It focuses on programming concepts and fundamentals within the context of solving real world problems.

Dr. Lenore Gervais Horowitz

This Programming for Problem Solving, by Lenore G. Horowitz, is copyrighted under the terms of a Creative Commons license:

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

For more information or feedback, contact:
Lenore G. Horowitz, Ph.D.
Information Science Department
University at Albany
State University of New York
1400 Washington Ave., Albany, NY 12222

Email: lhorowitz@albany.edu

Table of Contents

Preface 8
UNIT 1: How to Think Like an Engineer 11
Learning Objectives 11
Introduction 11
Computational Thinking 11
Algorithms 13
An Example Algorithm 14
Verifying your Algorithm 15
The Process of Computational Problem Solving 16
Values and Variables 17
What is a program? 18
Computational Problem Design using the Basic Programming Constructs 19
Sequential Control Example 20
Selection Control Examples 21
.... 21
Iterative Control Examples 23
The Role of Programming in the Field of Informatics 26
Unit Summary 27
Practice Problems 27
UNIT 2: Writing Simple Programs. 29
Learning Objectives 29
Computer Hardware Architecture 29
Digital Computing: It’s All about 0’s and 1’s 30
Operating Systems—Bridging Software and Hardware 31
Software Development Tools 32
Learning Programming with Python 35
Writing a Python Program 36
The Python Interactive Shell 39
Debugging 41
Syntax errors 41

Runtime errors 42
Semantic errors 42
Experimental debugging 42
The Basics of Python Programming 42
Comments 43
Literal Constants 43
Numbers 43
Strings 43
Variables 44
Identifier Naming 46
Indentation 47
Example: Using Variables and Literal Constants 48
Operators and Expressions 49
Operators 49
Practice with Operators & Expressions 51
Practice #1 51
Practice #2 52
Practice #3 52
Evaluation Order 52
Practice #4 - 53
Input/Process/Output Pattern 54
Type Converter Functions 55
Python's Standard Library 56
Math Module and Math Functions 56
More on Strings 57
Unit Summary 60
Practice Problems 60
UNIT 3: Graphics: Designing and developing graphics programs. 66
Learning Objectives 66
Object Oriented Programming 66
Using Objects 66
String Objects 67
Simple Graphics Programming 69

Graphics Windows: Coordinate Systems 72
GraphWin Objects 73
Example Program Using Coordinate Transformation 74
Graphics Objects 76
Point Methods 77
Line Methods 77
Circle Methods 77
Rectangle Methods 78
Oval Methods 78
Polygon Methods 78
Text Methods 78
Entry Objects 79
Displaying Images 80
Generating Colors 81
Interactive Graphics 82
Mouse Clicks 83
Handling Textual Input 85
Functions 88
Defining Functions 91
Function Parameters 92
The `return` Statement 94
Practice 95
Unit Summary 100
Practice Problems 100
UNIT 4: Control Structures: Making Decisions and Looping in computing. Data and Information Processing in Python. 102
Learning Objectives 102
Boolean Expressions 102
Logical operators 104
Conditional execution 104
Practice – 107
Exception Handling 110
Errors 110

Exceptions 111
Handling Exceptions 112
Practice 113
Practice with Handling Exceptions in our Programs 114
Practice #1 - Calculating the distance between two points: 114
Practice #2 – Guess the Number: 115
Iteration 117
The `while` Statement 117
Practice 1 118
Practice 2 120
Practice 3 122
The `break` Statement 123
The `continue` Statement 123
The `for...in` Statement 124
Practice 1 126
Practice 2 127
Nested Loops 129
Basic File Processing 131
Reading and Writing Text Files 132
Practice 134
Processing "Big Data" 135
Practice 136
Unit Summary 139
Practice Problems 139
Appendix A: Python on your Computer 142
Installing Python 3 and IDLE 142
Using Python and its IDE 143
File menu (Shell and Editor) 143
Edit menu (Shell and Editor) 144
Format menu (Editor window only) 145
Run menu (Editor window only) 145
Shell menu (Shell window only) 146
Debug menu (Shell window only) 146

Options menu (Shell and Editor) 146
Window menu (Shell and Editor) 147
Help menu (Shell and Editor) 147
Context Menus 147
Automatic indentation 148
Text colors 148

Table of Tables

Table 1: Compiler Vs Interpreter 34
Table 2: Python Keywords 47
Table 3: Python Operators 51
Table 4: Python `graphIntro.py` Code Explained 71
Table 5: Graphics Library Color Names 82
Table 6: Python Relationa/Comparisonl Operators 103
Table 7: File Functions and methods 132

Table of Figures

Figure 1: Computational Thinking (Big Deal Media) 12
Figure 2: Simple Algorithm (quora.com) 13
Figure 3: Example Algorithm 14
Figure 4: Testing Algorithms (securityintelligence.com) 15
Figure 5: Process of Computational Problem Solving 17
Figure 6: Python Code (businessinsider.com) 18
Figure 7: the 3 Programming Constructs 19
Figure 8: iPhone apps (abcnews.go.com) 26
Figure 9: The Digital Universe (https://www.emc.com/leadership/digital-universe/2014iview/images/du-stacks-to-moon.jpg) 26
Figure 10: Computer Hardware Architecture 29
Figure 11: Digital Representation 31
Figure 12: Operating Systems (wikipedia.org) 31
Figure 13: Listing of simple.py 37
Figure 14: New File Editor Window 37
Figure 15: Save Python File, Option 1 38
Figure 16: Save Python File, Option 2 38
Figure 17: Program simple.py Output 39
Figure 18: Executing Individual Commands in the Python Shell 40

Figure 19: First Computer Bug (Image ©Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988. NHHC Collection) 41
Figure 20: Input-Process-Output for Finding the Largest Number 54
Figure 21: Using the Built-in Function input() 54
Figure 22: Object Oriented Programming (wikibooks.org/wiki/File:CPT-Object-Var-Proc.svg) 67
Figure 23: List of String Methods 69
Figure 24: Example Graphics Window 71
Figure 25: Python Idle Shell Output 72
Figure 26: Default Graphics Coordinate System 72
Figure 27: Default Graphics Window 73
Figure 28: Match the Colors Game 74
Figure 29: Transformed Graphics Coordinate System 76
Figure 30: triangle.py 84
Figure 31: greeting.py Updated Window 86
Figure 32: greeting.py Opening Window 86
Figure 33: greeting.py with Coordinate Transformation 87
Figure 34:greetCoords.py 88
Figure 35: square.py 89
Figure 36: Conceptual view of the square root function 89
Figure 37: triangleGUIfunctions.py Graphics window output 99
Figure 38: Simple IF Conditional Flowchart 104
Figure 39: Alternative Execution Conditional Flowchart 105
Figure 40:Two or More Branches Conditional Flowchart 106
Figure 41: `if-elif-else` Conditional Flowchart 107
Figure 42: While Loop Flowchart 117
Figure 43: Sample Output from addNonNegatives.py 121
Figure 44: While Loop Flowchart 121
Figure 45: Sample Output from avgNonNegatives.py 122
Figure 46: Python Downloads Page 142

Preface

Much of this OER course material is from the following open publications:

1. Downey, Allen.Think Python: How to Think Like a Computer Scientist, 2nd Edition, Version 2.2.20. Green Tea Press, Needham, Massachusetts, 2015, http://greenteapress.com/wp/think-python-2e/ .

Permission is granted to copy, distribute, and/or modify this document under the terms of the Creative Commons Attribution-NonCommercial 3.0 Unported License, which is available at http: //creativecommons.org/licenses/by-nc/3.0/ .

2. 50 Examples for Teaching Python. A.M. Kuchling. Revision 2ef8c29a, 2012, http://fiftyexamples.readthedocs.io/en/latest/intro.html . Accessed 1 January 2018.

The English text of this work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/ .

3. Downey, Allen, et al. How to Think Like a Computer Scientist: Learning with Python. 2012, http://openbookproject.net/thinkcs/python/english3e/ .

Copyright (C) Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with Invariant Sections being Foreword, Preface, and Contributor List, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

4. Alvarado,Christine, et al. CS for All. Harvey Mudd College, 2013, https://www.cs.hmc.edu/csforall/.

Note: "Download or read it online for free."

5. Swaroop, C. H. "A Byte of Python." *Enllaç web,* 2003, https://python.swaroopch.com/ .

"A Byte of Python" is a free book on programming using the Python language. This book is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

This means:

You are free to Share i.e. to copy, distribute and transmit this book

You are free to Remix i.e. to make changes to this book (especially translations)

You are free to use it for commercial purposes

6. Brown, Wayne. "Lesson 6 – Introduction to Algorithmic Thinking." https://docs.google.com/document/d/1MyFYez2SQvsfq7r1StE2kxZyX.../edit . Accessed 3 January 2018.

Note: You can only view this document. To make changes, ask the owner for edit access.

7. Severance, Charles Russell, Sue Blumenberg, and Elliott Hauser. "Python for Everybody: Exploring Data in Python 3." (2016). https://www.py4e.com/book .

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. This license is available at http://creativecommons.org/licenses/by-nc-sa/3.0/

8. Halterman, R. "Fundamentals of Python Programming." Southern Adventist University (2017). http://python.cs.southern.edu/pythonbook/pythonbook.pdf

This document is copyright ©2017 by Richard L. Halterman, all rights reserved.

Permission is hereby granted to make hardcopies and freely distribute the material herein under the following conditions:

• The copyright and this legal notice must appear in any copies of this document made in whole or in part.

• None of material herein can be sold or otherwise distributed for commercial purposes without written permission of the copyright holder.

• Instructors at any educational institution may freely use this document in their classes as a primary or optional textbook under the conditions specified above.

A local electronic copy of this document may be made under the terms specified for hardcopies:

• The copyright and these terms of use must appear in any electronic representation of this document made in whole or in part.

• None of material herein can be sold or otherwise distributed in an electronic form for commercial purposes without written permission of the copyright holder.

• Instructors at any educational institution may freely store this document in electronic form on a local server as a primary or optional textbook under the conditions specified above.

I AM UNSURE ABOUT HOW TO CITE/USE THIS MATERIAL

9. *Python Software Foundation*. python, https://www.python.org/ . Accessed January 10, 2018.

Python is developed under an OSI-approved open source license, making it freely usable and distributable, even for commercial use. Python's license is administered by the Python Software Foundation.

10. Zelle, John M. "Graphics Module Reference." (2016).

Simple Graphics Library

The graphic library is released under the GPL, so it is freely available for use and modification.

graphics.py This is version 5.0 will work with both Python 2.x and 3.x. This is the latest version of the graphics library. It is known to work under Linux, Windows, and Mac OSX. Found at http://mcsp.wartburg.edu/zelle/python/graphics.py

Graphics Reference (HTML) Browseable reference documentation for the graphics package. Found at http://mcsp.wartburg.edu/zelle/python/graphics/graphics/index.html

Graphics Reference (PDF) *Downloadable/printable documentation for the graphics package. Found at http://mcsp.wartburg.edu/zelle/python/graphics/graphics.pdf*

11. Harrington, Andrew N. “Hands-on Python Tutorial (Python 3.1 Version).” Loyola University Chicago. https://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/index.html#

© Released under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License http://creativecommons.org/licenses/by-nc-sa/3.0/us/

12. Sarkis , Richard E. *CSC 161 Intro to Programming.* University of Rochester, 2018, http://www.pas.rochester.edu/~rsarkis/csc161/ . Accessed 25 February 2018.

This site is released under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License.

© Copyright 2018, Richard E Sarkis.

13. Jackson, Cody. *Learning to Program Using Python*. Publisher not identifed, 2014. Accessed 25 February 2018. https://www.ida.liu.se/~732A47/literature/PythonBook.pdf

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. Copyright © 2009-2013 Cody Jackson.

UNIT 1: How to Think Like an Engineer.

Learning Objectives

- Explain what we mean by “Computational Thinking”.
- Describe the problem being solved in a computational algorithm.
- Explain the process for generating computational algorithms.
- Generate and test algorithms to solve computational problems.
- Evaluate computational algorithms for exactness, correctness, termination, generalizability and understandability.
- Explain the role of programming in the field of Informatics.

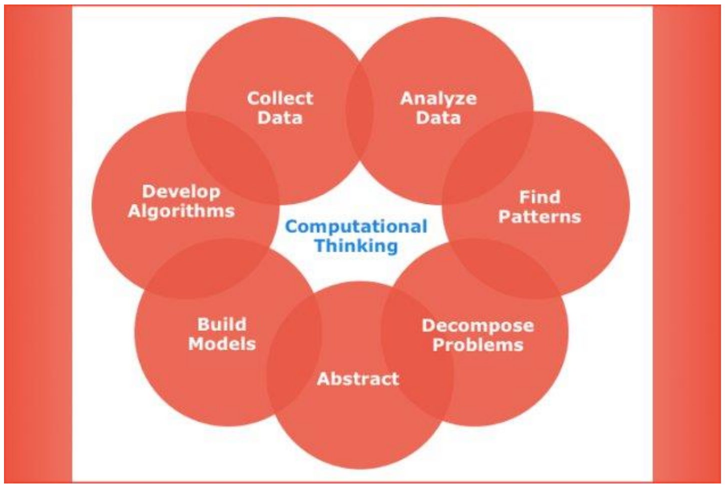
Introduction

The goal of this book is to teach you to solve computational problems and to think like an engineer. Computational problems are problems that can be solved by the use of computations (a computation is what you do when you calculate something). Engineers are people who solve problems - they invent, design, analyze, build and test "things" to fulfill objectives and requirements. The single most important skill for you to learn is problem solving. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills.

This book strives to prepare you to write well-designed computer programs that solve interesting problems involving data.

Computational Thinking

Computational Thinking is the thought processes involved in understanding a problem and expressing its solution in a way that a computer can effectively carry out. Computational thinking involves solving problems, designing systems, and understanding human behavior (e.g. what the user needs or wants) – thinking like an engineer. Computational thinking is a fundamental skill for everyone, not just for programmers because computational thinking is what comes *before* any computing technology.[1]



[1] Wing, Jeannette M. "Computational thinking." *Communications of the ACM* 49.3 (2006): 33-35.

Figure 1: Computational Thinking (Big Deal Media)

Computer science is the study of computation — what can be computed and how to compute it whereas computational thinking is:

The 6 parts of computational thinking...

> **Conceptualizing**, not programming. Computer science is not only computer programming. Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction;
>
> **Fundamental**, not rote skill. A fundamental skill is something every human being must know to function in modern society. Rote means a mechanical routine;
>
> **A way that humans, not computers, think**. Computational thinking is a way humans solve problems; it is not trying to get humans to think like computers. Computers are dull and boring; humans are clever and imaginative. We humans make computers exciting. Equipped with computing devices, we use our cleverness to tackle problems we would not dare take on before the age of computing and build systems with functionality limited only by our imaginations;
>
> **Complements and combines mathematical and engineering thinking**. Computer science inherently draws on mathematical thinking, given that, like all sciences, its formal foundations rest on mathematics. Computer science inherently draws on engineering thinking, given that we build systems that interact with the real world;
>
> **Ideas**, not artifacts. It's not just the software and hardware artifacts we produce that will be physically present everywhere and touch our lives all the time, it will be the computational concepts we use to approach and solve problems, manage our daily lives, and communicate and interact with other people;
>
> **For everyone, everywhere**. Computational thinking will be a reality when it is so integral to human endeavors it disappears as an explicit philosophy.[2]

[2] Wing, Jeannette M. "Computational thinking." *Communications of the ACM* 49.3 (2006): 33-35.

Algorithms

An algorithm specifies a series of steps that perform a particular computation or task. Throughout this book we'll examine a number of different algorithms to solve a variety of computational problems.

when you hear algorithm think recipes

algorithm = recipe

Algorithms resemble recipes. Recipes tell you how to accomplish a task by performing a number of steps. For example, to bake a cake the steps are: preheat the oven; mix flour, sugar, and eggs thoroughly; pour into a baking pan; set the timer and bake until done.

However, "algorithm" is a technical term with a more specific meaning than "recipe", and calling something an algorithm means that the following properties are all true:

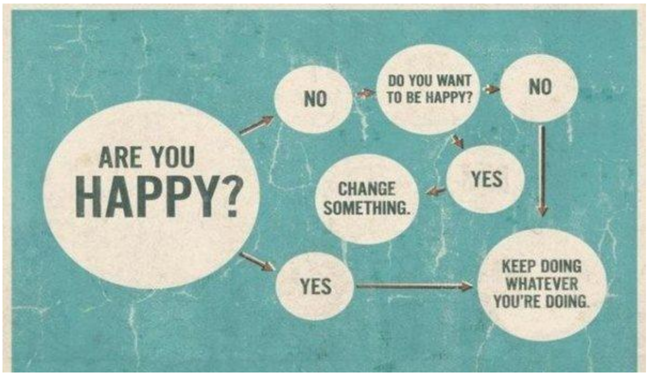


Figure 2: Simple Algorithm (quora.com)

Properties of an algorithm

1. An algorithm is an unambiguous description that makes clear what has to be implemented in order to solve the problem. In a recipe, a step such as "Bake until done" is ambiguous because it doesn't explain what "done" means. A more explicit description such as "Bake until the cheese begins to bubble" is better. In a computational algorithm, a step such as "Choose a large number" is vague: what is large? 1 million, 1 billion, or 100? Does the number have to be different each time, or can the same number be used again?

2. An algorithm expects a defined set of inputs. For example, it might require two numbers where both numbers are greater than zero. Or it might require a word, or a list customer names.

   An algorithm produces a defined set of outputs. It might output the larger of the two numbers, an all-uppercase version of a word, or a sorted version of the list of names.

3. An algorithm is guaranteed to terminate and produce a result, always stopping after a finite time. If an algorithm could potentially run forever, it wouldn't be very useful because you might never get an answer.

4. Must be general for any input it is given. Algorithms solve *general* problems (determine if a password is valid); they are of little use if they only solve a *specific* problem (determine if 'comp15' is a valid password).

5. It is at the right level of detail.....the person or device executing the instruction know how to accomplish the instruction without any extra information.

Once we know it's possible to solve a problem with an algorithm, a natural question is whether the algorithm is the best possible one. Can the problem be solved more quickly or efficiently?

Designing and algorithm

1. Understand completely the problem given

The first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully, then read it again. Try sketching out by hand some examples of how the problem can be solved. Finally consider any special cases and design your algorithm to address them.

2.Read the problems description carefully x2
3.Try sketching out examples of how to solve
4.consider any special cases, design to address them

An algorithm does not solve a problem rather it gives you a series of steps that, if executed correctly, will result in a solution to a problem.

An Example Algorithm

Let us look at a very simple algorithm called `find_max`.

Problem: Given a list of positive numbers, return the largest number on the list.

Inputs: A list of positive numbers. This list must contain at least one number. (Asking for the largest number in a list of no numbers is not a meaningful question.)

Outputs: A number, which will be the largest number in the list.

Algorithm:

```
1. Start
2. Accept a list of positive numbers; set to nums_list
3. Set max_number to 0.
3. For each number in nums_list, compare it to max_number.
    a. If the number is larger, set max_number to the larger number.
4. max_number is now set to the largest number in the list of positive numbers, nums_list.
5. End
```

Does this meet the criteria for being an algorithm?

- *Is it unambiguous?* Yes. Each step of the algorithm consists of uncomplicated operations, and translating each step into programming code is straight forward.
- *Does it have defined inputs and outputs?* Yes.
- *Is it guaranteed to terminate?* Yes. The list `nums_list` is of finite length, so after looking at every element of the list the algorithm will stop.
- *Is it general for any input?* Yes. A list of any set of positive numbers works.
- *Does it produce the correct result?* Yes. When tested, the results are what are expected.

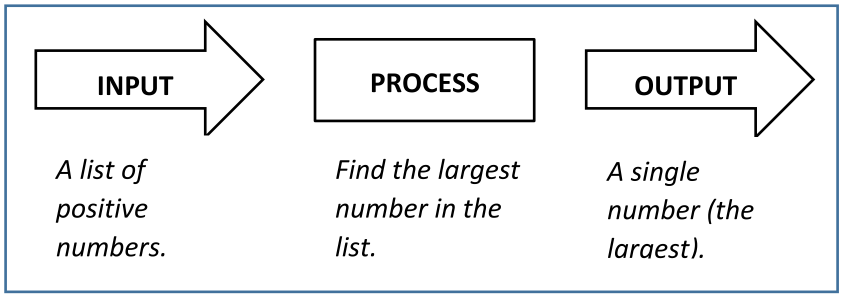

INPUT
PROCESS
OUTPUT
A list of positive numbers.
Find the largest number in the list.
A single number (the largest).

Figure 3: Example Algorithm

Verifying your Algorithm

How do we know if an algorithm is unambiguous, correct, comes to an end, is general AND is at the right level of detail? We must test the algorithm. *Testing* means verifying that the algorithm does what we expect it to do. In our 'bake a cake' example we know our algorithm is 'working' if, in the end, we get something that looks, smells and tastes like a cake.

Your first step should be to carefully read through EACH step of the algorithm to check for ambiguity and if there is any information missing. To ensure that the algorithm is correct, terminates and is general for any input we devise 'test cases' for the algorithm.

A test case is a set of inputs, conditions, and expected results developed for a particular computational problem to be solved. A test case is really just a question that you ask of the algorithm (e.g. if my list is the three numbers 2, 14, and 11 does the algorithm return the number 14?). The point of executing the test is to make sure the algorithm is correct, that it terminates and is general for any input.

Figure 4: Testing Algorithms (securityintelligence.com)

Good (effective) test cases:

- are easy to understand and execute
- are created with the user in mind (what input mistakes will be made? what are the preconditions?)
- make no assumptions (you already know what it is supposed to do)
- consider the boundaries for a specified range of values.

Let us look at the example algorithm from the previous section. The input for the algorithm is 'a list of positive numbers'. To make it easy to understand and execute keep the test lists short. The preconditions are that the list only contains numbers and these numbers must be positive so include a test with a 'non-number' (i.e. a special character or a letter) and a test with a negative number. The boundaries for the list are zero and the highest positive number so include a test with zero and a large positive number. That is it! Here is an example of three different test cases.

Test Case #	Input Values	Expected Result
1	List: 44, 14, 0, 1521, 89, 477	1521
2	List: 18, 4, 72, *, 31	Error (or no result)
3	List: 22, -9, 52	Error (or no result)

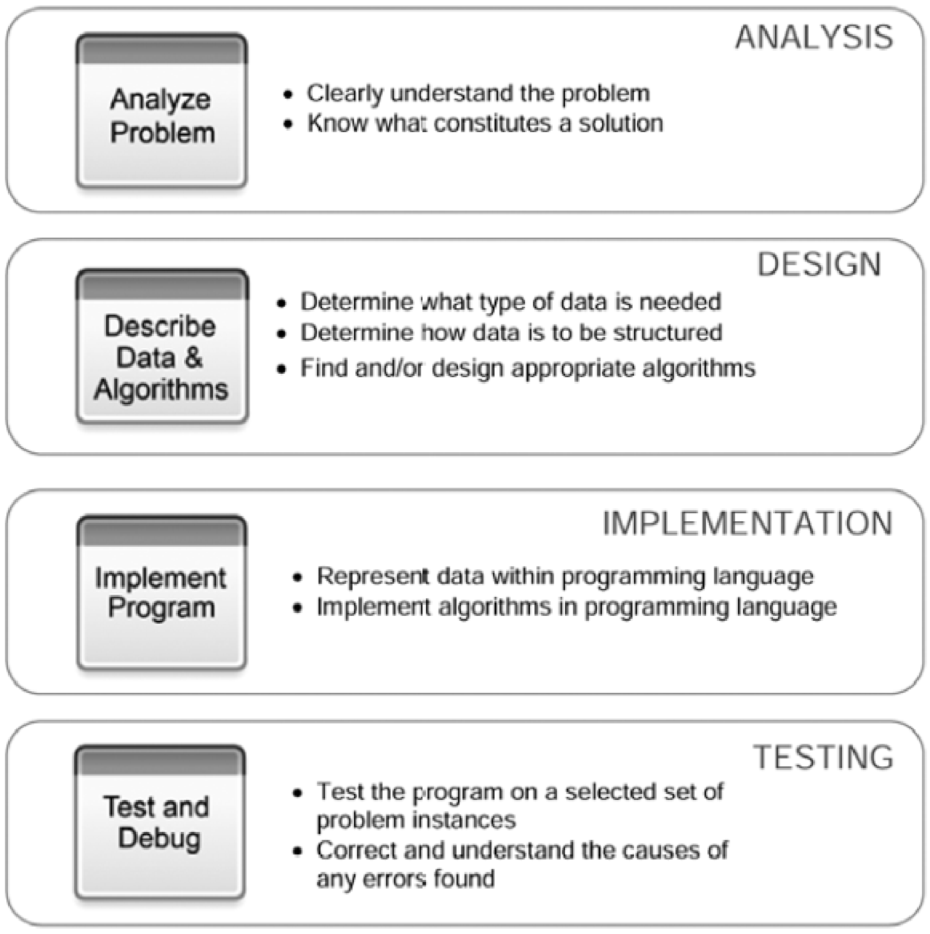
Manually, you should step through your algorithm using each of the three test cases, making sure that the algorithm does indeed terminate and that you get your expected result. As our algorithms and programs become more complex, skilled programmers often break each test case into individual steps of the algorithm/program and indicate what the expected result of each step should be. When you write a detailed test case, you don't necessarily need to specify the expected result for each test step if the result is obvious.

In computer programming we accept a problem to solve and develop an algorithm that can serve as a general solution. Once we have such a solution, we can use our computer to automate the execution. Programming is a skill that allows a competent programmer to take an algorithm and represent it in a notation (a program) that can be followed by a computer. These programs are written in **programming languages** (such as Python). Writing a correct and valid algorithm to solve a computational problem is key to writing good code. Learn to ***Think First*** and coding will come naturally!

The Process of Computational Problem Solving[3]

Computational problem solving does not simply involve the act of computer programming. It is a process, with programming being only one of the steps. Before a program is written, a design for the program must be developed (the algorithm). And before a design can be developed, the problem to be solved must be well understood. Once written, the program must be thoroughly tested. These steps are outlined in Figure 5.

[3] Dierbach, Charles. *Introduction to Computer Science Using Python: A Computational Problem-solving Focus*. Wiley Publishing, 2012, pp17-18.


ANALYSIS
Analyze Problem
Clearly understand the problem
Know what constitutes a solution
DESIGN
Describe Data & Algorithms
Determine what type of data is needed
Determine how data is to be structured
Find and/or design appropriate algorithms
IMPLEMENTATION
Implement Program
Represent data within programming language
Implement algorithms in programming language
TESTING
Test and Debug
Test the program on a selected set of problem instances
Correct and understand the causes of any errors found

> Steps to computational problem solving

Figure 5: Process of Computational Problem Solving

Values and Variables

A value is one of the basic things computer programs works with, like a password or a number of errors.

Values belong to different types: 21 is an integer (like the number of errors), and 'comp15' is a string of characters (like the password). Python lets you give names to *values* giving us the ability to generalize our algorithms.

One of the most powerful features of a programming language is the ability to use variables. A variable is simply a name that refers to a value as shown below,

`errors = 21`	variable `errors` is assigned the value `21`
`password = 'comp15'`	variable `password` is assigned the value `'comp15'`

Whenever the variable `errors` appears in a calculation the current value of the variable is used.

`errors = 21`	variable `errors` is assigned the value `21`
`errors = errors + 1`	variable `errors` is assigned the value `of 21 +1 (22)`

We need some way of storing information (i.e. the number of errors or the password) and manipulate them as well. This is where variables come into the picture. Variables are exactly what the name implies - their value can vary, i.e., you can store anything using a variable. Variables are just parts of your computer's memory where you store some information. Unlike literal constants, you need some method of accessing these variables and hence you give them names.

Programmers generally choose names for their variables that are meaningful and document what the variable is used for. It is a good idea to begin variable names with a lowercase letter . The underscore character (_) can appear in a name and is often used in names with multiple words.

What is a program?

A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of mathematical equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or something graphical, like processing user input on an ATM device.

```
        _user(self, user):
    Returns a QuerySet of connections for user.
    """
    set1 = self.filter(from_user=user).select_related(depth=1)
    set2 = self.filter(to_user=user).select_related(depth=1)
    return set1 | set2
def are_connected(self, user1, user2):
    if self.filter(from_user=user1, to_user=user2).count() > 0:
        return True
    if self.filter(from_user=user2, to_user=user1).count() > 0:
        return True
    return False
f remove(self, user1, user2):
    """
    Deletes proper object regardless of the order of users in argument
    """
    connection = self.filter(from_user=user1, to_user=user2)
    if not connection:
        connection = self.filter(from_user=user2, to_user
    connection.delete()
odels.py        Top
```

Figure 6: Python Code (businessinsider.com)

The details look different in different computer programming languages, but there are some low-level conceptual patterns (constructs) that we use to write all programs. These constructs are not just for Python programs, they are a part of every programming language.

input Get data from the “outside world”. This might be reading data from a file, or even some kind of sensor like a microphone or GPS. In our initial algorithms and programs, our input will come from the user typing data on the keyboard.

Conceptual Patterns/Constructs used to write programs <

output Display the results of the program on a screen or store them in a file or perhaps write them to a device like a speaker to play music or speak text.

sequential execution Perform statements one after another in the order they are encountered in the script.

conditional execution Checks for certain conditions and then executes or skips a sequence of statements.

repeated execution Perform some set of statements repeatedly, usually with some variation.

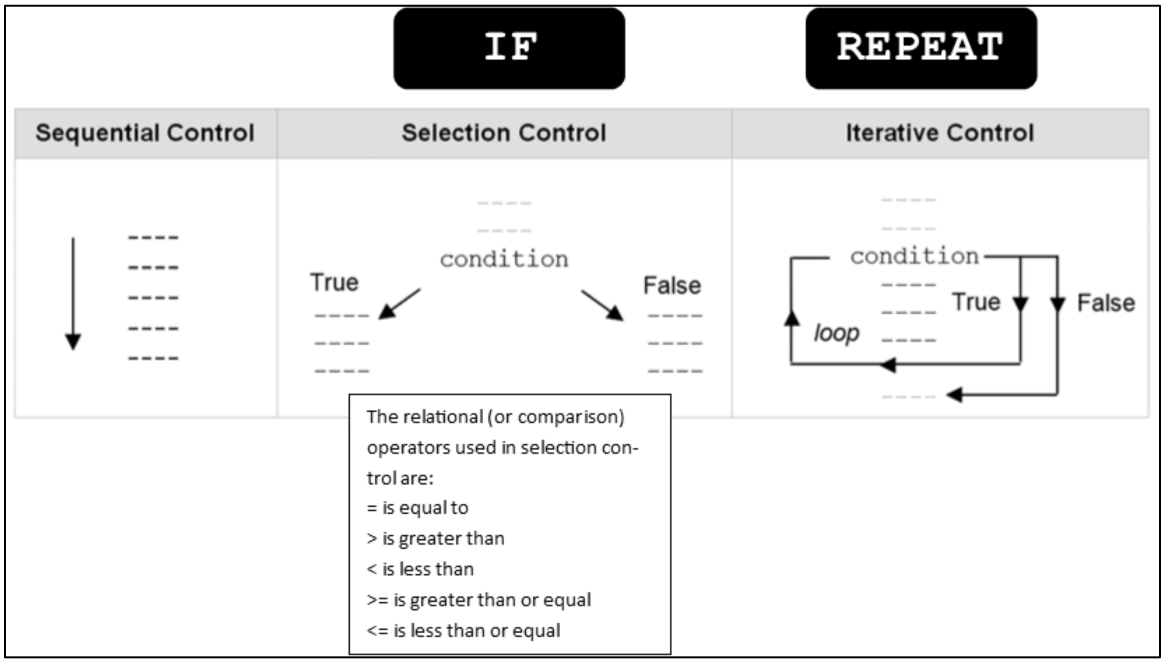
reuse Write a set of instructions once and give them a name and then reuse those instructions as needed throughout your program.

Believe it or not, that's pretty much all there is to it. Every computer application you've ever used, no matter how complicated, is made up of constructs that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic constructs. The "art" of writing a program is composing and weaving these basic elements together many times over to produce something that is useful to its users.

Computational Problem Design using the Basic Programming Constructs

The key to better algorithm design and thus to programming lies in limiting the **control structure** to only three constructs as shown below.

1. The Sequence structure (sequential execution)
2. The Decision, Selection or Control structure (conditional execution)
3. Repetition or Iteration Structure (repeated execution)



The relational (or comparison) operators used in selection control are:
= is equal to
> is greater than
< is less than
>= is greater than or equal
<= is less than or equal

Figure 7: the 3 Programming Constructs[4]

Let us look at some examples for the sequential control and the selection control.

[4] Based on a figure from: Dierbach, Charles. *Introduction to Computer Science Using Python: A Computational Problem-solving Focus*. Wiley Publishing, 2012, pp81.

Sequential Control Example

The following algorithm is an example of **sequential control**.

Problem: Given two numbers, return the sum and the product of the two numbers.
Inputs: Two numbers.
Outputs: The sum and the product.

```
1.Start
2.display "Input two numbers"
3.accept number1, number2
4.sum = number1 + number2
5.print "The sum is ", sum
6.product = number1 * number2
7.print "The product is ", product
8.End
```

Does this meet the criteria for being an algorithm?

- *Is it unambiguous?* Yes. Each step of the algorithm consists of uncomplicated operations, and translating each step into programming code is straight forward.
- *Does it have defined inputs and outputs?* Yes.
- *Is it guaranteed to terminate?* Yes. Sequential control, by its nature, always ends.
- *Is it general for any input?* Yes. Any two numbers work in this design.
- *Does it produce the correct result?* Yes. When tested, the results are what are expected.

Here is an example of three different test cases that are used to verify the algorithm.

Test Case #	Input Values	Expected Result
1	numbers 0 and 859	sum is 859 product is 0
2	numbers -5 and 10	sum is 5 product is -50
3	numbers 12 and 3	sum is 15 product is 36

Selection Control Examples

The following two algorithms are examples of **selection control** which uses the 'IF' statement in most programming languages.

Problem: Given two numbers, the user chooses to either multiply, add or subtract the two numbers. Return the value of the chosen calculation.

Inputs: Two numbers and calculation option.

Outputs: The value of the chosen calculation.

```
1.start
2.display "choose one of the following"
3.display "m for multiply"
4.display "a for add"
5.display "s for subtract"
6.accept choice
7.display "input two numbers you want to use"
8.accept number1, number2
9.if choice = m then answer= number1 * number2
10.if choice = a then answer= number1 + number2
11.if choice = s then answer= number1 -number2
12. if choice is not m, a, or s then answer is NONE
12.display answer
13.stop
```

The relational (or comparison) operators used in selection control are:

=	is equal to
>	is greater than
<	is less than
>=	is greater than or equal
<=	is less than or equal
<>	is not equal to

Does this meet the criteria for being an algorithm?

- *Is it unambiguous?* Yes. Each step of the algorithm consists of uncomplicated operations, and translating each step into programming code is straight forward.
- *Does it have defined inputs and outputs?* Yes.
- *Is it guaranteed to terminate?* Yes. The input is of finite length, so after accepting the user's choice and the two numbers the algorithm will stop.
- *Is it general for any input?* Yes. Any two numbers work in this design and only a choice of a'm', 'a', or 's' will result in numeric output.
- *Does it produce the correct result?* Yes. When tested, the results are what are expected.

Here is an example of three different test cases that are used to verify the algorithm.

Test Case #	Input Values	Expected Result
1	choice ‘a’ numbers -12 and 32	answer is 20 terminate
2	choice ‘s’ numbers -2012 and 0	answer is 2012 terminate
3	choice ‘**’ numbers 8 and 4	answer is NONE terminate

This example uses an extension of the simple selection control structure we just saw and is referred to as the ‘IF-ELSE’ structure.

<u>Problem</u>: Accept from the user a positive integer value representing a salary amount, return tax due based on the salary amount.

<u>Inputs</u>: One positive integer number.

<u>Outputs</u>: The calculated tax amount.

```
1.start

2.accept salary

3.If salary < 50000 then

4.Tax = 0 Else

5.If salary > 50000 AND salary < 100000 then

6.Tax = 50000 * 0.05 Else

7.Tax = 100000 * 0.30

8.End IF

9.display Tax
```

The relational (or comparison) operators used in selection control are:

= is equal to

> is greater than

< is less than

>= is greater than or equal

<= is less than or equal

<> is not equal to

Does this meet the criteria for being an algorithm?

- *Is it unambiguous?* Yes. Each step of the algorithm consists of uncomplicated operations, and translating each step into programming code is straight forward.
- *Does it have defined inputs and outputs?* Yes.
- *Is it guaranteed to terminate?* Yes. The input is of finite length, so after accepting the user’s number, even if it is negative, the algorithm will stop.
- *Is it general for any input?* Yes. Any number entered in this design will work.

- *Does it produce the correct result?* Yes. When tested, the results are what are expected.

Here is an example of three different test cases that are used to verify the algorithm.

Test Case #	Input Values	Expected Result
1	salary of 0	tax is 0 terminate
2	salary of 75000	tax is 2500 terminate
3	salary of 120000	tax is 30000 terminate

Iterative Control Examples

The third programming control is the iterative or, also referred to as, the repetition structure. This control structure causes certain steps to be repeated in a sequence a specified number of times or until a condition is met. This is what is called a 'loop' in programming

Indefinite Loops and Definite Loops

In all programming languages there are generally two options: an indefinite loop (the Python 'WHILE' programming statement) and a definite loop (the Python 'FOR' programming statement). We can use these two constructs, WHILE and FOR, for iterations or loops in our algorithms.

Note for Reader:
A **definite loop** is where we know exactly the number of times the loop's body will be executed. Definite iteration is usually best coded as a Python `for` loop.
An **indefinite loop** is where we do not know before entering the body of the loop the exact number of iterations the loop will perform. The loop just keeps going until some condition is met. A `while` statement is used in this case.

The following algorithm is an example of **iterative control** using **WHILE**.

Problem: Print each keyboard character the users types in until the user chooses the 'q' (for 'quit') character.
Inputs: A series of individual characters.
Outputs: Each character typed in by the user.

```
1.Start
2.initialize (set) letter = 'a'
3.WHILE letter <> 'q'
4.ACCEPT letter
5.DISPLAY "The character you typed is", letter
6.end WHILE
7.Stop
```

Does this meet the criteria for being an algorithm?

- *Is it unambiguous?* Yes. Each step of the algorithm consists of uncomplicated operations, and translating each step into programming code is straight forward.
- *Does it have defined inputs and outputs?* Yes.
- *Is it guaranteed to terminate?* Yes. The input is of finite length, so after accepting the user's keyboard character, even if it is not a letter, the algorithm will stop.
- *Is it general for any input?* Yes. Any keyboard character entered in this design will work.
- *Does it produce the correct result?* Yes. When tested, the results are what are expected.

Here is an example of three different test cases that are used to verify the algorithm.

Test Case #	**Input Values**	**Expected Result**
1	letter 'z'	The character you typed is z. Ask for another letter.
2	letter '8'	The character you typed is 8 Ask for another letter.
3	letter 'q'	The character you typed is q. Terminate.

The following algorithm is an example of **iterative control** using **FOR**. This statement is used when the number of iterations is known in advance.

Problem: Ask the user how many words they want to enter then print the words entered by the user.
Inputs: Number of words to be entered; this value must be a positive integer greater than zero. Individual words.
Outputs: Each word typed in by the user.

```
1.Start
2.accept num_words (must be at least one)
3.repeat num_words times (FOR 1 to num_words)
4. accept word
5. DISPLAY "The word you entered is", word
6.end FOR
```

Does this meet the criteria for being an algorithm?

- *Is it unambiguous?* Yes. Each step of the algorithm consists of uncomplicated operations, and translating each step into programming code is straight forward.
- *Does it have defined inputs and outputs?* Yes.
- *Is it guaranteed to terminate?* Yes. The input is of finite length, so after accepting the user's number of words to enter and any characters typed on the keyboard, even if it is not a 'word' per say, the algorithm will stop.
- *Is it general for any input?* Yes. Any positive integer greater than zero and any size 'word' will work.
- *Does it produce the correct result?* Yes. When tested, the results are what are expected.

Here is an example of two different test cases that are used to verify the algorithm.

Test Case #	**Input Values**	**Expected Result**
1	num_words 1 word 'code'	The word you entered is 'code'. Terminate.
2	num_words 3 word 'coding' word 'is' word 'fun'	The word you entered is 'coding'. Ask for another word. The word you entered is 'is'. Ask for another word. The word you entered is 'fun'. Terminate.

Note to Reader:
An **infinite loop** is a loop that executes its block of statements repeatedly until the user forces the program to quit. Once the program flow enters the loop's body it cannot escape. Infinite loops sometimes are by design. For example, a long-running server application like a Web server may need to continuously check for incoming connections. The Web server can perform this checking within a loop that runs indefinitely.

Beginning programmers, unfortunately, all too often create infinite loops by accident, and these infinite loops represent logic errors in their programs.

The Role of Programming in the Field of Informatics

You see computer programming in use every day. When you use Google or your smartphone, or watch a movie with special effects, there is programing at work. When you order a product over the Internet, there is code in the web site, in the cryptography used to keep your credit card number secure, and in the way that UPS routes their delivery vehicle to get your order to you as quickly as possible.

Figure 8: iPhone apps (abcnews.go.com)

Programming is indeed important to an informatics professional as they are interested in finding solutions for a wide variety of computational problems involving data.

When you Google the words "pie recipe," Google reports that it finds approximately 38 million pages, ranked in order of estimated relevance and usefulness. Facebook has approximately 1 billion active users who generate over 3 billion comments and "Likes" each day. GenBank, a national database of DNA sequences used by biologists and medical researchers studying genetic diseases, has over 100 million genetic sequences with over 100 billion DNA base pairs. According to the International Data Corporation, by 2020 the digital universe – the data we create and copy annually – will reach 44 zettabytes, or 44 trillion gigabytes.

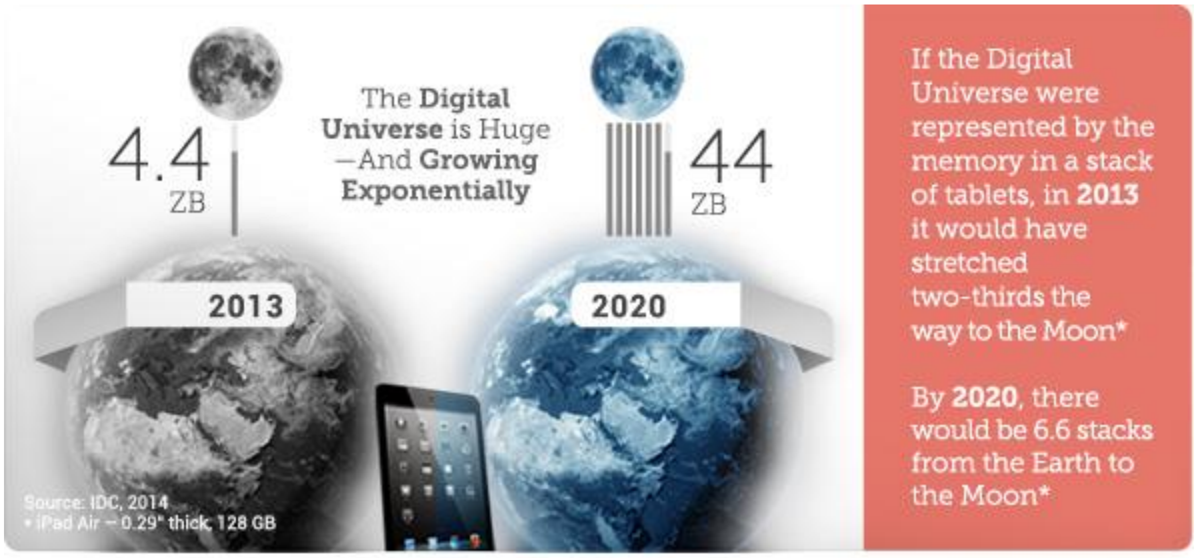

4.4 ZB
The Digital Universe is Huge —And Growing Exponentially
44 ZB
2013
2020
If the Digital Universe were represented by the memory in a stack of tablets, in 2013 it would have stretched two-thirds the way to the Moon*
By 2020, there would be 6.6 stacks from the Earth to the Moon*
Source: IDC, 2014
* iPad Air – 0.29" thick, 128 GB

Figure 9: The Digital Universe (www.emc.com/leadership/digital-universe/2014iview/images)

Doing meaningful things with data is challenging, even if we're not dealing with millions or billions of things. In this book, we will be working with smaller sets of data. But much of what we'll do will be applicable to very large amounts of data too.

Unit Summary

Computational Thinking is the thought processes involved in formulating a problem and expressing its solution in a way that a computer—human or machine—can effectively carry out.

Computational Thinking is what comes before any computing technology—thought of by a human, knowing full well the power of automation.

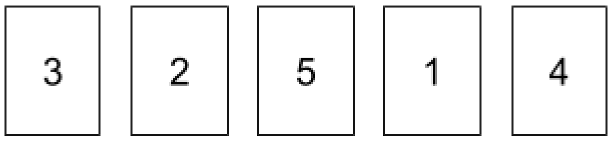
Writing a correct and valid algorithm to solve a computational problem is key to writing good code.

1. First, understand the problem
    a. *What are the inputs?*
    b. *What are the outputs (or results)?*
    c. *Can we break the problem into parts?*
2. Second, design the process (write the **algorithm**!).
    a. *Think about the connections between the input & output.*
    b. *Consider designing 'backwards'.*
    c. *Have you seen the problem before? In a slightly different form?*
    d. *Can you solve part of the problem?*
    e. *Did you use all the inputs?*
3. Third, test your design.
    a. *Can you test it on a variety of inputs?*
    b. *Can you think of how you might write the algorithm differently if you had to start again?*
4. Last, write the program (we will be addressing this in our next Unit).
    a. *Does it solve the problem? Does it meet all the requirements? Is the output correct?*
    b. *Does it terminate?*
    c. *Is it general for all cases?*

Practice Problems

1. Write about a process in your life (e.g. driving to the mall, walking to class, etc.) and estimate the number of steps necessary to complete the task. Would you consider this a complex or simple task? What happens if you scale that task (e.g. driving two states away to the mall)? Is your method the most efficient? Can you come up with a more efficient way?

2. Given five cards randomly placed in a row like the ones below write an algorithm to sort the cards. How would your algorithm change if there were 10 cards? What about 100 cards?



3. Write an algorithm to find the average of 25 test grades out of a possible 100 points.

4. If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, it is clear that you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle: “If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can.”
   Write an algorithm that accepts three integers as arguments, and that displays either “Yes” or “No,” depending on whether you can or cannot form a triangle from sticks with the given lengths.

5. ROT13 is a weak form of encryption that involves “rotating” each letter in a word by 13 places. To rotate a letter means to shift it through the alphabet, wrapping around to the beginning if necessary, so ‘A’ shifted by 3 is ‘D’ and ‘Z’ shifted by 1 is ‘A’.
   Write an algorithm that accepts a word and an integer from the user, and that prints a new encrypted word that contains the letters from the original word “rotated” by the given amount (the integer input). For example, “cheer” rotated by 7 is “jolly” and “melon” rotated by −10 is “cubed.”

6. Write an algorithm to accept a score between 0.0 and 1.0. If the score is out of range, display an error message. If the score is between 0.0 and 1.0, display a grade using the following table:

Score	Grade
>= 0.9	A
>= 0.8	B
>= 0.7	C
>= 0.6	D
< 0.6	E

7. Write an algorithm which repeatedly accepts numbers until the user enters “done”. Once “done” is entered, display the total sum of all the numbers, the count of numbers entered, and the average of all the numbers.

8. Write an algorithm that sums a series of ten positive integers entered by the user excluding all numbers greater than 100. Display the final sum.

UNIT 2: Writing Simple Programs.

Learning Objectives

- Explain the dependencies between hardware and software
- Describe the form and the function of computer programming languages
- Create, modify, and explain computer programs following the input/process/output pattern.
- Form valid Python identifiers and expressions.
- Write Python statements to output information to the screen, assign values to variables, and accept information from the keyboard.
- Read and write programs that process numerical data and the Python math module.
- Read and write programs that process textual data using built-in functions and methods.

Computer Hardware Architecture

Before we start learning a programming language to give instructions to computers to develop software, we need to learn about how computers are built. If you were to take apart your computer or cell phone and look deep inside, you would find the following parts:

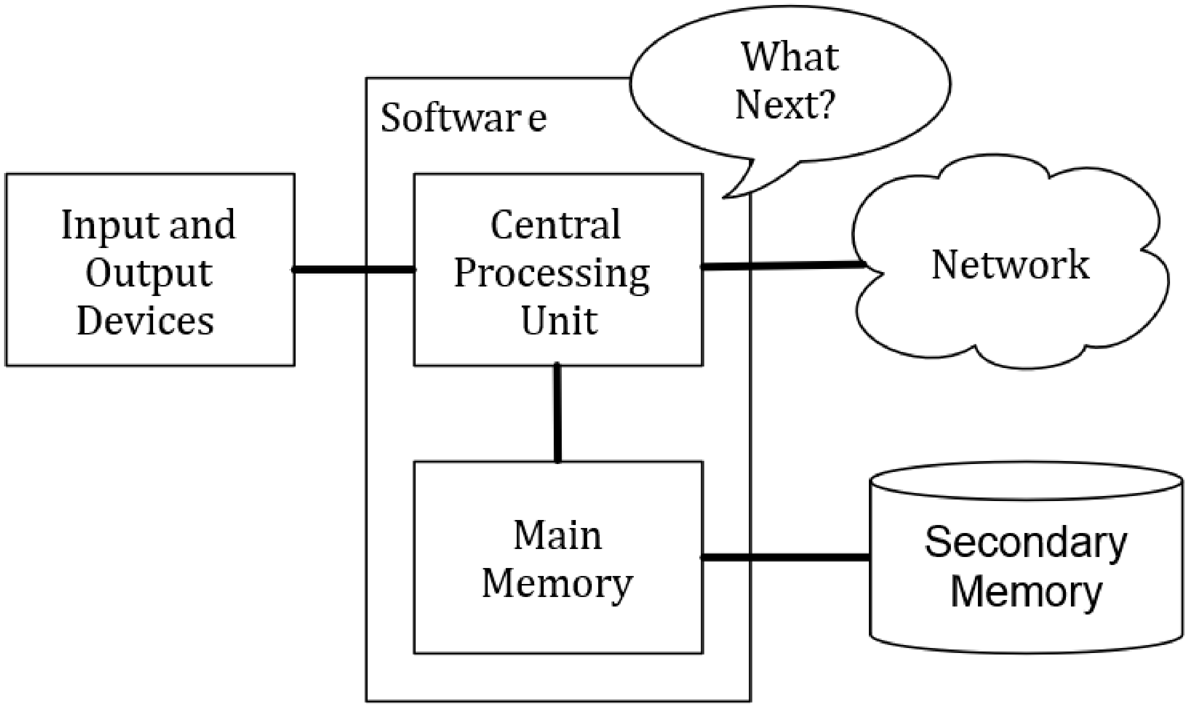


Figure 10: Computer Hardware Architecture

The high-level definitions of these parts are as follows:

- The *Central Processing Unit* (or CPU) is the part of the computer that is built to be obsessed with "what is next?" If your computer is rated at 3.0 Gigahertz, it means that the CPU will ask "What next?" three billion times per second.
- The *Main Memory* is used to store information that the CPU needs in a hurry. The main memory is nearly as fast as the CPU. But the information stored in the main memory vanishes when the computer is turned off.
- The *Secondary Memory* is also used to store information, but it is much slower than the main memory. The advantage of the secondary memory is that it can store information even when there is no power to the computer. Examples of secondary memory are disk drives or flash memory (typically found in USB sticks and portable music players).
- The *Input and Output Devices* are simply our screen, keyboard, mouse, microphone, speaker, touchpad, etc. They are all of the ways we interact with the computer.
- These days, most computers also have a *Network Connection* to retrieve information over a network. We can think of the network as a very slow place to store and retrieve data that might not always be "up". So in a sense, the network is a slower and at times unreliable form of *Secondary Memory*.

While most of the detail of how these components work is best left to computer builders, it helps to have some terminology so we can talk about these different parts as we write our programs.

As a programmer, your job is to use and orchestrate each of these resources to solve the problem that you need to solve and analyze the data you get from the solution. As a programmer you will mostly be "talking" to the CPU and telling it what to do next. Sometimes you will tell the CPU to use the main memory, secondary memory, network, or the input/output devices.

Digital Computing: It’s All about 0’s and 1’s

It is essential that computer hardware be reliable and error free. If the hardware gives incorrect results, then any program run on that hardware is unreliable. The key to developing reliable systems is to keep the design as simple as possible[5]. In digital computing, all information is represented as a series of digits or electronic symbols which are either "on" or "off" (similar to a light switch). These patterns of electronic symbols are best represented as a sequence of zeroes and ones, digits from the binary (base 2) number system.

[5] Dierbach, Charles. *Introduction to Computer Science Using Python: A Computational Problem-solving Focus*. Wiley Publishing, 2012, p9.

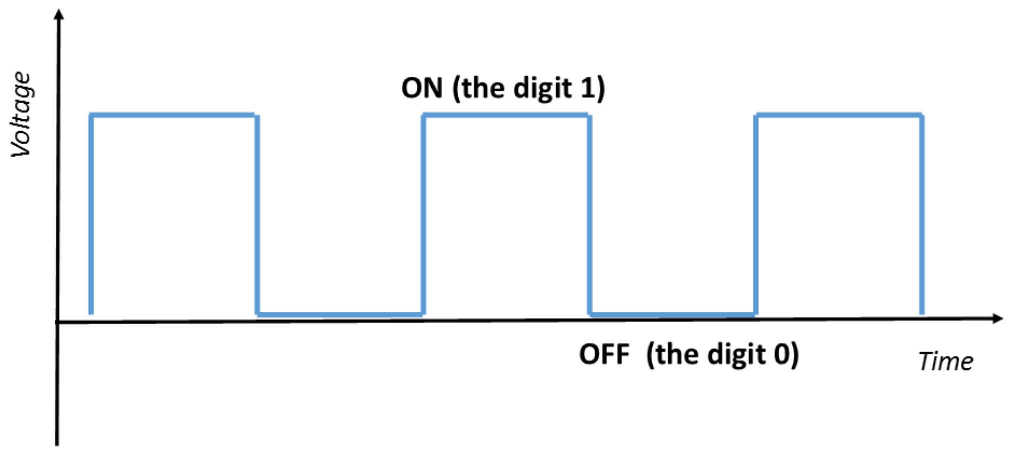


Figure 11: Digital Representation

The term bit stands for binary digit. Therefore, every bit has the value 0 or 1. A byte is a group of bits operated on as a single unit in a computer system, usually consisting of eight bits. Although values represented in base 2 are significantly longer than those represented in base 10, binary representation is used in digital computing because of the resulting simplicity of hardware design[6]. For example, the decimal number `485` is represented in binary as `111100101`.

Operating Systems—Bridging Software and Hardware[7]

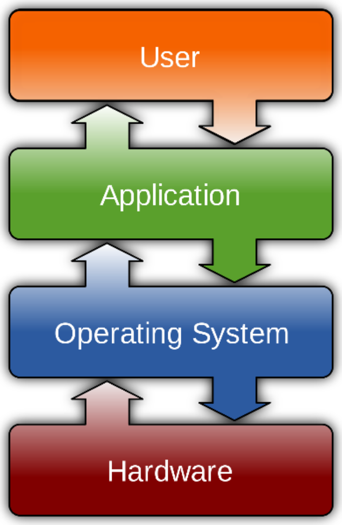


Figure 12: Operating Systems (wikipedia.org)

An operating system is software that has the job of managing and interacting with the hardware resources of a computer. Because an operating system is intrinsic to the operation a computer, it is referred to as system software.

An operating system acts as the "middle man" between the hardware and executing application programs (see Figure 12). For example, it controls the allocation of memory for the various programs that may be executing on a computer. Operating systems also provide a particular user interface. Thus, it is the operating system installed on a given computer that determines the "look and feel" of the user interface and how the user interacts with the system, and not the particular model computer.

[6] Dierbach, Charles. *Introduction to Computer Science Using Python: A Computational Problem-solving Focus*. Wiley Publishing, 2012, p10

[7] Dierbach, Charles. *Introduction to Computer Science Using Python: A Computational Problem-solving Focus*. Wiley Publishing, 2012, pp.11-12.

Software Development Tools

It can be very cumbersome, error prone, and time consuming to converse with a computer using only zeros and ones. Numerical machine code (computer code using only zeros and ones) does exist but is rarely used by programmers. For that reason most people program using a "higher-level" programming language which use words and symbols that are easier for humans to manage than binary sequences. Tools exist that automatically convert a higher-level description of what is to be done into the required lower-level, machine code. Higher-level programming languages like Python allow programmers to express solutions to programming problems in terms that are much closer to a natural language like English. Some examples of the more popular of the hundreds of higher-level programming languages that have been devised over the past 60 years include FORTRAN, COBOL, Lisp, Haskell, C, Perl, C++, Java, and C#. Most programmers today, especially those concerned with high-level applications, usually do not worry about the details of underlying hardware platform and its machine language.

Sample numerical machine code:

```
0010100011101001001010100000001111
111001100000111010100101011011 01
...
```

Fortunately, higher-level programming languages provide a relatively simple structure with very strict rules for forming statements, called the programming language syntax, which can express a solution to any problem that can be solved by a computer.

Consider the following program fragment written in the Python programming language:

```
subtotal = 25
tax = 3
total = subtotal + tax
```

While these three lines (three statements) do constitute a proper Python program, they are more likely a small piece of a larger program. The lines of text in this program fragment look similar to expressions in algebra. We see no sequence of binary digits. Three words, `subtotal`, `tax`, and `total`, called variables, represent information. In programming, a variable represents a value stored in the computer's memory. Instead of some cryptic binary instructions meant only for the CPU, we see familiar-looking mathematical operators (= and +). Since this program is expressed in the Python language, not machine language, no computer processor (CPU) can execute the program directly. A program called an **interpreter** translates the Python code into machine code when a user runs the program. The higher-level language code is called the source code. The corresponding machine language code is called the target code. The interpreter translates the source code into the target machine language.

The beauty of higher-level languages is this: the same Python source code can execute on different target platforms. The target platform must have a Python interpreter available, but multiple Python interpreters are available for all the major computing platforms. The human programmer therefore is free to think about writing the solution to the problem in Python, not in a specific machine language.

Programmers have a variety of tools available to enhance the software development process. Some common tools include:

- **Editors**. An editor allows the programmer to enter the program source code and save it to files. Most programming editors increase programmer productivity by using colors to highlight language features. The syntax of a language refers to the way pieces of the language are arranged to make well-formed "sentences." To illustrate, the sentence

  The tall boy runs quickly to the door.

  uses proper English syntax. By comparison, the sentence

  Boy the tall runs door to quickly the.

  is not correct syntactically. It uses the same words as the original sentence, but their arrangement does not follow the rules of English.

  Similarly, programming languages have strict syntax rules that programmers must follow to create well-formed programs. Only well-formed programs are acceptable for translation into executable machine code. Some syntax-aware editors can use colors or other special annotations to alert programmers of syntax errors during the editing process.

- **Compilers**. A compiler translates the source code to target code. The target code may be the machine language for a particular platform or embedded device. The target code could be another source language; for example, the earliest C++ compiler translated C++ into C, another higher-level language. The resulting C code was then processed by a C compiler to produce an executable program. (C++ compilers today translate C++ directly into machine language.) Compilers translate the contents of a source file and produce a file containing all the target code. Popular compiled languages include C, C++, Java, C#.

- **Interpreters**. An interpreter is like a compiler, in that it translates higher-level source code into target code (usually machine language). It works differently, however. While a compiler produces an executable program that may run many times with no additional translation needed, an interpreter translates source code statements into machine language each time a user runs the program. A compiled program does not need to be recompiled to run, but an interpreted program must be reinterpreted each time it executes. The interpreter in essence reads the code one line at a time. In general, compiled programs execute more quickly than interpreted programs because the translation activity occurs only once. Interpreted programs, on the other hand, can run as is on any platform with an appropriate interpreter; they do not need to be recompiled to run on a different platform. Python, for example, is used mainly as an interpreted language, but compilers for it are available. Interpreted languages are better suited for dynamic, explorative development which many people feel is ideal for beginning programmers. Popular scripting languages include Python, Ruby, Perl, and, for web browsers, Javascript.

  The Python interpreter is written in a high-level language called "C". You can look at the actual source code for the Python interpreter by going to www.python.org and working your way to their source code. So Python is a program itself and it is compiled into machine code. When you install Python on your computer, you copy a machine-code copy of the translated Python program onto your system. In Windows, the executable machine code for Python itself is likely in a file with a name like:

```
C:\Python35\python.exe
```

- **Debuggers**. A debugger allows a programmer to more easily trace a program's execution in order to locate and correct errors in the program's implementation. With a debugger, a developer can simultaneously run a program and see which line in the source code is responsible for the program's current actions. The programmer can watch the values of variables and other program elements to see if their values change as expected. Debuggers are valuable for locating errors (also called bugs) and repairing programs that contain errors. (See the Debugging Section in this Unit for more information about programming errors.)

COMPILER	INTERPRETER
Compiler takes an entire program as input. It works on the complete program at once.	Interpreter takes a single statement at a time as input. It works line by line.
It generates Intermediate Object code (machine code).	It doesn't generate Intermediate code (machine code).
It executes Conditional control statements faster than Interpreter. It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.	It executes Conditional control statements much slower than Compiler. In general, the overall execution time is slower.
More memory required (since Object Code is generated).	Memory requirement is less, hence more memory efficient. It doesn't generate intermediate Object Code.
Compiled program doesn't need to be compiled every time.	Every time higher level program is converted into lower level program.
Errors are displayed after entire program is checked. Hence debugging is comparatively hard.	Errors are displayed for every instruction interpreted (if any). Continues translating the program until the first error is met, in which case it stops. Hence debugging is easier.
Programming languages that use compilers are COBOL, C, C++.	Programming languages that use interpreter are Visual Basic Script, Ruby, Perl.

Table 1: Compiler Vs Interpreter[8]

Many developers use integrated development environments (IDEs). An IDE includes editors, debuggers, and other programming aids in one comprehensive program. Python IDEs include Wingware, PyCharm,, and IDLE.

Despite the wide variety of tools (and tool vendors' claims), the programming process for all but trivial programs is not automatic. Good tools are valuable and certainly increase the productivity of

[8] Based on information from: https://www.softwaretestingmaterial.com/compiler-vs-interpreter/

developers, but they cannot write software. There are no substitutes for sound logical thinking, creativity, common sense, and, of course, programming experience.

Learning Programming with Python

Guido van Rossum created the Python programming language in the late 1980s. He named the language after the BBC show "Monty Python's Flying Circus". In contrast to other popular languages such as C, C++, Java, and C#, Python strives to provide a simple but powerful syntax.

Python is used for software development at companies and organizations such as Google, Yahoo, Facebook, CERN, Industrial Light and Magic, and NASA. It is especially relevant in developing information science applications such as ForecastWatch.com which uses Python to help meteorologists, online travel sites, airline reservation systems, university student record systems, air traffic control systems among many others[9]. Experienced programmers can accomplish great things with Python, but Python's beauty is that it is accessible to beginning programmers and allows them to tackle interesting problems more quickly than many other, more complex languages that have a steeper learning curve.

Python has an extensive Standard Library which is a collection of built-in modules, each providing specific functionality beyond what is included in the "core" part of Python. (For example, the `math` module provides additional mathematical functions. The `random` module provides the ability to generate random numbers)[10]. Additionally the Standard Library can help you do various things involving regular expressions, documentation generation, databases, web browsers, CGI, FTP, email, XML, HTML, WAV files, cryptography, GUI (graphical user interfaces), among other things.

More information about Python, including links to download the latest version for Microsoft Windows, Mac OS X, and Linux, can be found in Appendix A of this book and also at http://www.python.org .

In late 2008, Python 3.0 was released. Commonly called Python 3, the current version of Python, VERSION 3.0, is incompatible with earlier versions of the language. Many existing books and online resources cover Python 2, but more Python 3 resources now are becoming widely available. The code in this book is based on Python 3.

This book does not attempt to cover all the facets of the Python programming language. The focus here is on introducing programming techniques and developing good habits and skills. To that end, this approach avoids some of the more obscure features of Python and concentrates on the programming basics that easily transfer directly to other programming languages.

The goal is to turn you into a person who is skilled in the art of programming. In the end you will be a programmer - perhaps not a professional programmer, but at least you will have the skills to look at a data/information analysis problem and develop a program to solve the problem.

[9] www.python.org/about/success/

[10] Dierbach, Charles. *Introduction to Computer Science Using Python: A Computational Problem-solving Focus*. Wiley Publishing, 2012, p23

In a sense, you need two skills to be a programmer:

- First, you need to know the programming language (Python) - you need to know the vocabulary and the grammar (the syntax). You need to be able to spell the words in this new language properly and know how to construct well-formed "sentences" in this new language.
- Second, you need to "tell a story". In writing a story, you combine words and sentences to convey an idea to the reader. There is a skill and art in constructing the story, and skill in story writing is improved by doing some writing (practice) and getting some feedback. In programming, our program is the "story" and the problem you are trying to solve is the "idea".

Once you learn one programming language such as Python, you will find it much easier to learn a second programming language such as JavaScript or C++. Other programming languages have very different vocabulary and grammar (syntax) but the problem-solving skills will be the same across all programming languages.

You will learn the "vocabulary" and "sentences" (the syntax) of Python pretty quickly. It will take longer for you to be able to write a coherent program to solve a new problem. We learn programming much like we learn writing. We start by reading and explaining programs, then we write simple programs, and then we write increasingly complex programs over time. At some point you "get your muse" and see the patterns on your own and can see more naturally how to take a problem and write a program that solves that computational problem. And once you get to that point, programming becomes a very pleasant and creative process.

We start with the vocabulary and structure of Python programs. Be patient as the simple examples remind you of when you started reading for the first time.

Writing a Python Program

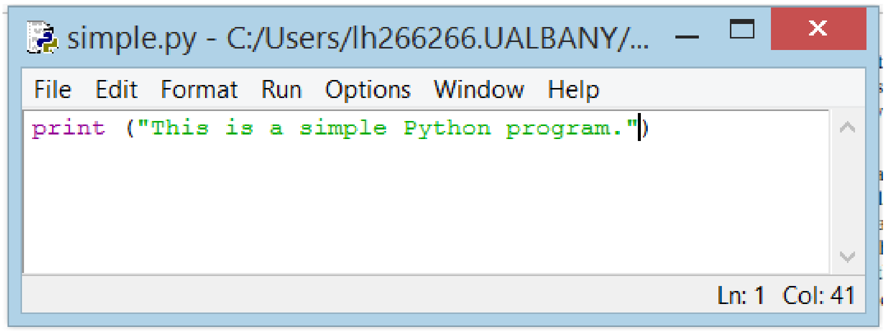
The text that makes up a Python program has a particular structure. The syntax must be correct, or the interpreter will generate error messages and not execute the program. This section introduces Python by providing a simple example program.

A program consists of one or more **statements**. A statement is an instruction that the interpreter executes.

The following statement invokes the `print` function to display a message:

```
print("This is a simple Python program")
```

We can use the statement in a program. Figure 13 (`simple.py`) is an example of a very simple Python program that does something:


simple.py - C:/Users/lh266266.UALBANY/...
File Edit Format Run Options Window Help
Ln: 1 Col: 41

```
print ("This is a simple Python program.")
```

Figure 13: Listing of simple.py

IDLE is Python's Integrated Development and Learning Environment (IDE) and is included as part of the Python Standard Library which is distributed with Python 3 (See Appendix A). IDLE is the standard Python development environment. Its name is an acronym of "**I**ntegrated **D**eve**L**opment **E**nvironment". It works well on both Unix and Windows platforms.

IDLE has a **Python Shell window**, which gives you access to the Python interactive mode. It also has a file editor that lets you create and edit existing Python source files. The file editor was used to write the `simple.py` program.

The way you launch IDLE depends on your operating system and how it was installed. Figure 13 shows a screenshot of IDLE running on a Windows 8.1 computer. The IDE consists of a simple menu bar at the top. Other Python IDEs are similar in appearance.

To begin entering our program, we just type in the Python statements. To start a new program, choose the **New File** item from the **File** menu. This action produces a new editor pane for a file named `Unititled` as shown in Figure 14 below.

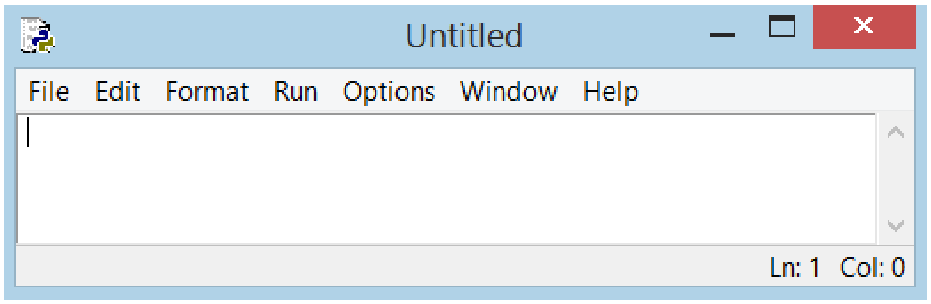
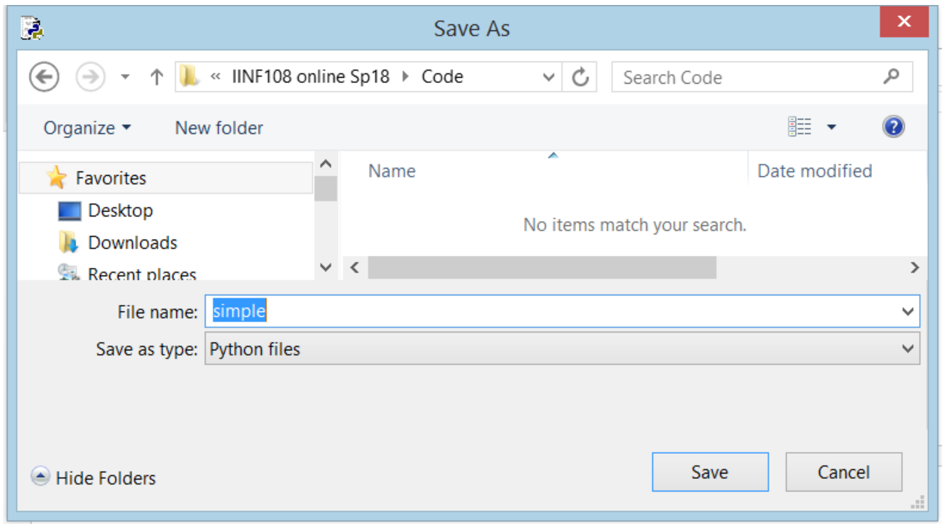


Figure 14: New File Editor Window

We now are ready to type in the code that constitutes the program.

```
print("This is a simple Python program")
```

Next we will save the file. The menu sequence **File→Save** or **File→ Save As**, produces the dialog box shown in Figure 15 that allows us to select a folder and filename for our program. You should be sure all Python programs are saved with a `.py` extension. If "Save as Type: Python files" is available there is no need to add the `.py` extension as it will automatically be saved as a `.py` file (see Figure 15).



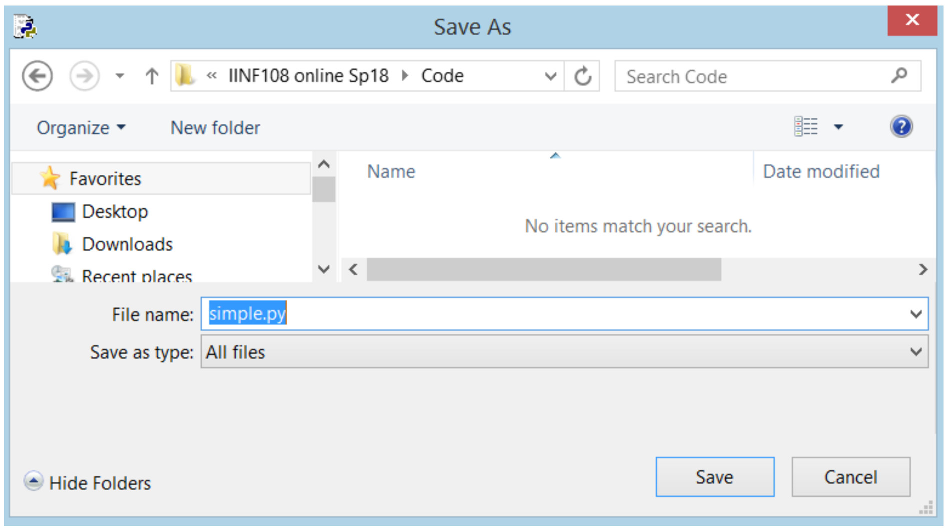
No items match your search.

File name: simple

Save as type: Python files

Figure 15: Save Python File, Option 1

If you are using a different text editor select “Save as Type: All files” and add the `.py` extension (see Figure 16).



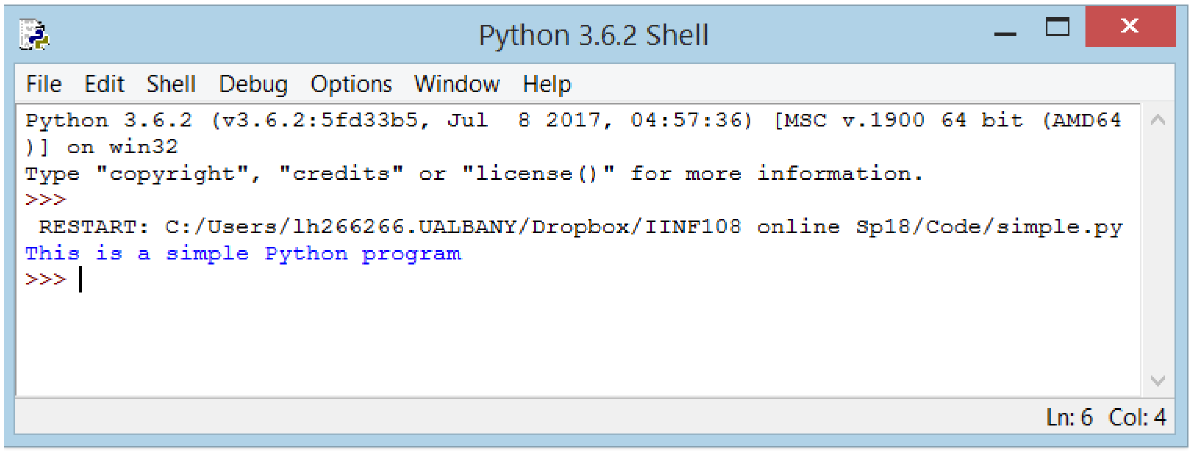
No items match your search.

File name: simple.py

Save as type: All files

Figure 16: Save Python File, Option 2

We can run the program by selecting the menu sequence **Run→Run Module** or use the shortcut key **F5**. A new window labeled `Python Shell` will display the program’s output. Figure 17 shows the results of running the program.



```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64
)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 
 RESTART: C:/Users/lh266266.UALBANY/Dropbox/IINF108 online Sp18/Code/simple.py 
This is a simple Python program
>>> |
```

Figure 17: Program simple.py Output

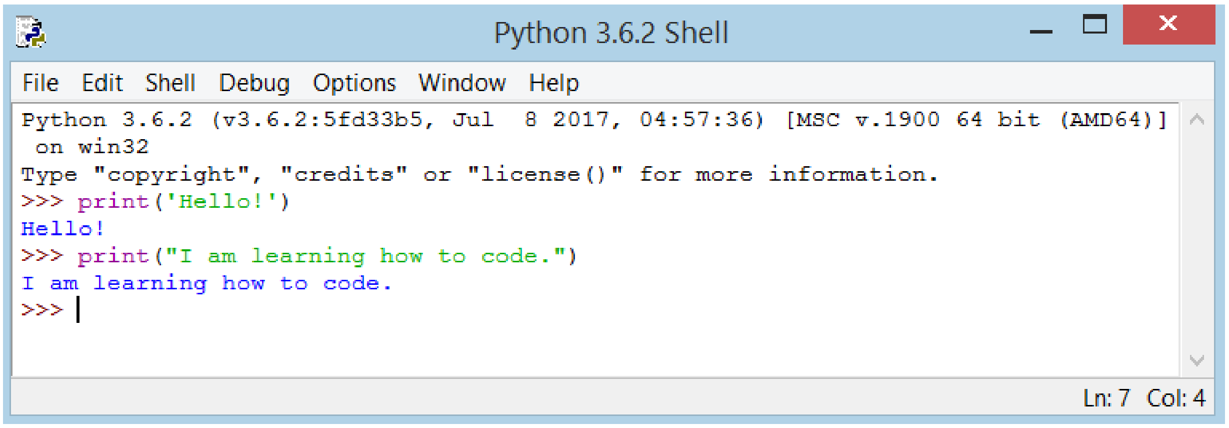
This program contains one Python statement. A statement is a command that the interpreter executes. This statement prints the message `This is a simple Python program` in the Python Shell window. A statement is the fundamental unit of execution in a Python program. Statements may be grouped into larger chunks called blocks, and blocks can make up more complex statements (e.g. the selection structure or the iterative structure we saw in the last Unit). The statement `print("This is a simple Python program")` makes use of a built in function named `print`. Python has a variety of different kinds of statements that we can use to build programs, and the sections that follow explore these various kinds of statements.

Note to Reader: In the context of programming, a function is a named sequence of statements that performs a computation. The name of the function here is `print`. The expression in parentheses is called the **argument** of the function. The result, for this function, is the string of characters in quotes (i.e. the 'message') of the **argument**.

When you type a statement on the command line in the Shell window and select the Enter key, Python executes it. Statements alone don't produce any result.

The Python Interactive Shell

We created the program in Figure 13 (`simple.py`) and submitted it to the Python interpreter for execution. We can interact with the interpreter directly, typing in Python statements and expressions for immediate execution. As we saw in Figure 17, the IDLE window labeled *Python Shell* is where the executing program directs its output. We also can type commands into the Python Shell window, and the interpreter will attempt to execute them. Figure 18 shows how the interpreter responds when we enter the program statement directly into the Shell. The interpreter prompts the user for input with three greater-than symbols (>>>). This means the user typed in the text on the line prefixed with >>>. Any lines without the >>> prefix represent the interpreter's output, or feedback, to the user. We will find Python's interactive interpreter invaluable for experimenting with various language constructs.



```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>> print('Hello!')
Hello!
>>> print("I am learning how to code.")
I am learning how to code.
>>> 
```

Figure 18: Executing Individual Commands in the Python Shell

We can discover many things about Python without ever writing a complete program. We can execute the interactive Python interpreter directly from the command line in the Python Shell. The interpreter prompt (>>>) prefixes all user input in the interactive Shell. Lines that do not begin with the >>> prompt represent the interpreter's response.

If you try to enter each line one at a time into the interactive Shell, the program's output will be intermingled with the statements you type. In this case the best approach is to type the program into an editor, save the code you type to a file, and then execute the program. Most of the time we use an editor to enter and run our Python programs. The interactive interpreter is most useful for experimenting with small snippets of Python code.

Note to Reader:

The `print()` function always ends with an invisible "new line" character (\n) so that repeated calls to print will all print on a separate line each. To prevent this newline character from being printed, you can specify that it should end with a blank:

```
print('a', end='')
print('b', end='')
```

Output is:

```
ab
```

Or you can end with a space:

```
print('a', end=' ')
print('b', end=' ')
print('c')
```

Output is:

```
a b
```

Or you can end with a space:

```
print('a', end=' ')
print('b', end=' ')
print('c')
```

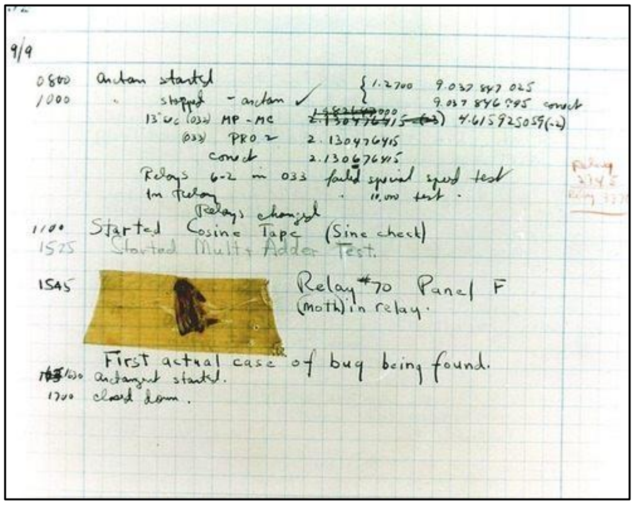
Output is:

```
a b c
```

Debugging

Programming is a complex process, and because it is done by human beings, it often leads to errors. Programming errors are called bugs and the process of tracking them down and correcting them is called debugging.

The story behind this term dates back to September 9, 1947, when Harvard's Mark II Aiken Relay computer was malfunctioning. After rooting through the massive machine to find the cause of the problem, Admiral Grace Hopper, who worked in the Navy's engineering program at Harvard, found the bug. It was an actual insect. The incident is recorded in Hopper's logbook alongside the offending moth, taped to the logbook page: "15:45 Relay #70 Panel F (moth) in relay. First actual case of bug being found."[11]



9/9

0800 Antan started {1.2700 9.037 847 025

1000 " stopped - antan ✓ 9.037 846 795 correct

13"uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (-3) 4.615925059(-2)

(033) PRO 2 2.130476415

correct 2.130676415

Relays 6-2 in 033 failed special speed test

In relay . 11,000 test .

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult+ Adder Test.

1545 Relay #70 Panel F

(moth) in relay.

First actual case of bug being found.

1630 Antangent started.

1700 closed down.

Figure 19: First Computer Bug (Image ©Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988. NHHC Collection)

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

Syntax errors

Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. Syntax refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period.

this sentence contains a syntax error.

So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of E. E. Cummings without problems. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will display an error message and quit, and you will not be able to run your program. Early on, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

[11] Jacobson, Molly McBride. "Grace Hopper's Bug." Atlas Obscura, https://www.atlasobscura.com/places/grace-hoppers-bug . Accessed January 9, 2018.

Runtime errors

The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first Units, so it might be a while before you encounter one.

Semantic errors

The third type of error is the semantic error. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages and quit, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing. The test cases we generated in UNIT #1 helps programmers fix semantic errors.

Experimental debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one.

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does something and make small modifications, debugging them as you go, so that you always have a working program.

The Basics of Python Programming

Just printing a single sentence is not enough, is it? You want to do more than that - you want to take some input, manipulate it and get something out of it. We can achieve this in Python using constants and variables, and we'll learn some other concepts as well in this section.

Comments

Comments are any text to the right of the **#** symbol and is mainly useful as notes for the reader of the program.

For example:

```
print('hello world') # Note that print is a function

      or:

# Note that print is a function
print('hello world')
```

Use as many useful comments as you can in your program to:

- explain assumptions
- explain important decisions
- explain important details
- explain problems you're trying to solve
- explain problems you're trying to overcome in your program, etc.

Code tells you **how**, comments should tell you **why**.

This is useful for readers of your program so that they can easily understand what the program is doing. Consider the programmer, newly hired, who has been assigned the job of maintaining a 2000 line-of-code program. Without comments the codes can be very difficult to understand let alone maintain.

Literal Constants

An example of a literal constant is a number like `5`, `1.23`, or a string like `'This is a string'` or `"It's a string!"` (String literals must be in quotes).

It is called a literal because it is literal - you use its value literally. The number 2 always represents itself and nothing else - it is a constant because its value cannot be changed. Hence, all these are referred to as literal constants.

Numbers

Numbers are mainly of two types - integers and floats. An example of an integer is `2` which is just a whole number. Examples of floating point numbers (or floats for short) are `3.23` and `7845.322222`.

Strings

A string is a sequence of characters. Strings can be a single character, a single word or a bunch of words. You will be using strings in many Python programs that you write. Note the following:

Single Quote: You can specify (define) strings using single quotes such as `'Quote me on this'`. All white space i.e. spaces and tabs, within the quotes, are preserved as-is.

Double Quotes: Strings in double quotes work exactly the same way as strings in single quotes. An example is `"What's your name?"` .There is no difference in using single quotes or double quotes , just be sure to use a matching set.

Triple Quotes: You can specify multi-line strings using triple quotes - (""" or '''). You can use single quotes and double quotes freely within the triple quotes. An example is:

```
'''This is a multi-line string. This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."
'''
```

Strings are immutable. This means that once you have created a string, you cannot change it. Although this might seem like a bad thing, it really isn't. We will see why this is not a limitation in the various programs that we see later on.

Variables

Using just literal constants can soon become boring - we need some way of storing any information and manipulate them as well. This is where variables come into the picture. Variables are exactly what the name implies - their value can vary, i.e., you can store anything using a variable. Variables are just parts of your computer's memory where you store some information. Unlike literal constants, you need some method of accessing these variables and hence you give them names.

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value.

The **assignment statement** gives a value to a variable:

```
>>> message = "What is today’s date?"
>>> n = 17
>>> pi = 3.14159
```

This example makes three assignments. The first assigns the string value `"What is today’s date?"` to a variable named `message`. The second assigns the integer 17 to `n`, and the third assigns the floating-point number 3.14159 to a variable called `pi`.

The **assignment symbol**, =, should not be confused with equals, which uses the symbol ==. The assignment statement binds a name, on the left-hand side of the operator, to a value, on the right-hand side. This is why you will get an error if you enter:

```
>>> 17 = n
File "<interactive input>", line 1
SyntaxError: can't assign to literal
```

Tip: When reading or writing code, say to yourself "**n** is assigned 17" or "**n** gets the value 17".

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of diagram is called a state snapshot because it shows what state each of the variables is in at a particular instant in time. (Think of it as the variable's state of mind). The following diagram shows the result of executing the assignment statements:

```
message → "What is today's date?"
      n → 17
     pi → 3.14159
```

If you ask the interpreter to evaluate a variable in the Python Shell, it will produce the value that is currently linked to the variable:

```
>>> message
'What is today's date?'
>>> n
17
>>> pi
3.14159
```

We use variables in a program to "remember" things, perhaps the current score at the football game. But variables are variable. This means they can change over time, just like the scoreboard at a football game. You can assign a value to a variable, and later assign a different value to the same variable. (This is different from mathematics. In mathematics, if you give `x` the value 3, it cannot change to link to a different value half-way through your calculations!). For example:

```
>>> day = "Thursday"
>>> day
'Thursday'
>>> day = "Friday"
>>> day
'Friday'
>>> day = 21
>>> day
21
```

You'll notice we changed the value of day three times, and on the third assignment we even made it refer to a value that was of a different data type.

A great deal of programming is about having the computer remember things, e.g. The number of missed calls on your phone, and then arranging to update or change the variable when you miss another call.

Note to Reader:

There are two common variable genres that are used in computer programming. They are so regularly used that they have special names.

accumulator: A variable used in a loop to add up or accumulate a result.

counter: A variable used to count something, usually initialized to zero and then incremented.

Identifier Naming

Variables are examples of identifiers. Identifiers are names given to identify something. There are some rules you have to follow for naming identifiers:

- The first character of the identifier must be a letter of the alphabet (uppercase ASCII or lowercase ASCII or Unicode character) or an underscore (_).
- The rest of the identifier name can consist of letters (uppercase ASCII or lowercase ASCII or Unicode character), underscores (_) or digits (0-9).
- Identifier names are case-sensitive. For example, myname and myName are not the same. Note the lowercase n in the former and the uppercase N in the latter.
- Examples of valid identifier names are `i` , `name_2_3` . Examples of invalid identifier names are `2things`, `this is spaced out`, `my-name` and `>a1b2_c3`

Python keywords define the language's syntax rules and structure, and they cannot be used as variable names.

Python has thirty-something keywords (and every now and again improvements to Python introduce or eliminate one or two):

`and`	`as`	`assert`	`break`	`class`	`continue`
`def`	`del`	`elif`	`else`	`except`	`exec`
`finally`	`for`	`from`	`global`	`if`	`import`
`in`	`is`	`lambda`	`nonlocal`	`not`	`or`
`pass`	`raise`	`return`	`try`	`while`	`with`
`yield`	`True`	`False`	`None`		

Table 2: Python Keywords

You might want to keep this table handy. If the interpreter complains about one of your variable names and you don't know why, see if it is in this table.

Programmers generally choose names for their variables that are meaningful to the human readers of the program — they help the programmer document, or remember, what the variable is used for.

Note to Reader:

Beginners sometimes confuse "meaningful to the human readers" with "meaningful to the computer". So they'll wrongly think that because they've called some variable `average` or `pi`, it will somehow magically calculate an average, or magically know that the variable `pi` should have a value like 3.14159. No! The computer doesn't understand what you intend the variable to mean.

So you'll find some textbooks or tutorials that deliberately don't choose meaningful names when they teach beginners — not because we don't think it is a good habit, but because we're trying to reinforce the message that you — the programmer — must write the program code to calculate the average, and you must write an assignment statement to give the variable pi the value you want it to have.

Indentation

Whitespace is important in Python. Actually, whitespace at the beginning of the line is important. This is called indentation. Leading whitespace (spaces and tabs) at the beginning of the logical line is used to determine the indentation level of the logical line, which in turn is used to determine the grouping of statements.

This means that statements which go together must have the same indentation. Each such set of statements is called a block. We will see examples of how blocks are important in later sections and Units.

One thing you should remember is that wrong indentation can give rise to errors. For example:

```
i = 5
# Error below! Notice a single space at the start of the line
 print('Value is', i)
print('I repeat, the value is', i)
```

When you run this, you get the following error:

```
File "whitespace.py", line 3
 print('Value is', i)
^
IndentationError: unexpected indent
```

Notice that there is a single space at the beginning of the second line. The error indicated by Python tells us that the syntax of the program is invalid i.e. the program was not properly written. What this means to you is that you cannot arbitrarily start new blocks of statements (except for the default main block which you have been using all along, of course). Cases where you can use new blocks (such as the iteration control structure) will be detailed in later Units.

How to indent: Use four spaces for indentation. This is the official Python language recommendation. Good editors (including IDLE) will automatically do this for you. Make sure you use a consistent number of spaces for indentation, otherwise your program will not run or will have unexpected behavior.

Example: Using Variables and Literal Constants

Practice: Type, save and run the following program, `var.py`, using the Python editor.

```
# Filename : var.py
i = 5
print(i)
i = i + 1
print(i)
s = '''This is a multi-line string.
This is the second line.'''
print(s)
```

Output:

```
5
6
This is a multi-line string.
This is the second line.
```

Let us examine how this program works.

Python Statement	Explanation
`i = 5`	First, we assign the literal constant value `5` to the variable `i` using the assignment operator (`=`). This line is called a statement because it states that something should be done and in this case, we connect the variable name `i` to the value `5`.
`print(i)`	Next, we print the value of `i` using the print statement which, unsurprisingly, just prints the value of the variable to the screen
`i = i + 1`	Here we add 1 to the value stored in `i` and store it back into `i`.
`print(i)`	We then print it and expectedly, we get the value `6` .

`s = '''This is a multi-line string.`
`This is the second line.'''`
	Here we assign the literal string constant to the variable `s` .

`print(s)`	We then print it.

Operators and Expressions

Most statements (logical lines) that you write will contain expressions. A simple example of an expression is `2 + 3`. An expression can be broken down into operators and operands.

Operators are functionality that do something and can be represented by symbols such as + or by special keywords. Operators require some data to operate on and such data is called operands. In this case, `2` and `3` are the operands.

When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed.

Operators

We will briefly take a look at the operators and their usage.

Note that you can evaluate the expressions given in the examples using the interpreter interactively. For example, to test the expression 2 + 3 , use the interactive Python interpreter prompt in the Shell window:

```
>>> 2 + 3
5
>>> 3 * 5
15
>>>
```

Here is a quick overview of the available operators:

+ (plus)	Adds two objects	3 + 5 gives 8 . 'a' + 'b' gives 'ab' .
- (minus)	Gives the subtraction of one number from the other; if the first operand is absent it is assumed to be zero.	-5.2 gives a negative number 50 - 24 gives 26 .
* (multiply)	Gives the multiplication of the two numbers or returns the string repeated that many times.	2 * 3 gives 6 . 'la' * 3 gives 'lalala' .
** (power)	Returns x to the power of y	
(divide)	Divide x by y	13 / 3 gives 4.333333333333333
// (divide and floor)	Divide x by y and round the answer down to the nearest whole number	13 // 3 gives 4 -13 // 3 gives -5
% (modulo)	Returns the remainder of the division	13 % 3 gives 1 . -25.5 % 2.25 gives 1.5 .
< (less than)	Returns whether x is less than y. All comparison operators return True or False . Note the capitalization of these names.	5 < 3 gives False 3 < 5 gives True . Comparisons can be chained arbitrarily: 3 < 5 < 7 gives True .
> (greater than)	Returns whether x is greater than y	5 > 3 returns True . If both operands are numbers, they are first converted to a common type. Otherwise, it always returns False .

<= (less than or equal to)	Returns whether x is less than or equal to y	x = 3;
		y = 6;
		x <= y returns True

>= (greater than or equal to)	Returns whether x is greater than or equal to y	x = 4;
		y = 3;
		x >= 3 returns True

== (equal to)	Compares if the objects are equal	x = 2; y = 2; x == y returns True
		x = 'str'; y = 'stR'; x == y returns False
		x = 'str'; y = 'str'; x == y returns True

!= (not equal to)	Compares if the objects are not equal	x = 2;
		y = 3;
		x != y returns True

not (boolean NOT)	If x is True , it returns False . If x is False , it returns True .	x = True; not x returns False .
and (boolean AND)	x and y returns False if x is False , else it returns evaluation of y	x = False; y = True; x and y returns False since x is False. In this case, Python will not evaluate y since it knows that the left hand side of the 'and' expression is False which implies that the whole expression will be False irrespective of the other values. This is called short-circuit evaluation.

or (boolean OR)	If x is True , it returns True, else it returns evaluation of y	x = True; y = False; x or y returns True . Short-circuit evaluation applies here as well.

Table 3: Python Operators

Some of the comparison operators work on strings. For example the + operator (plus) works with strings, but it is not addition in the mathematical sense. Instead it performs **concatenation**, which means joining the strings by linking them end to end.

The `*` operator also works on strings; it performs repetition. For example, `'Fun'*3` is `'FunFunFun'`. One of the operands has to be a string; the other has to be an integer.

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25
>>> first = '100'
>>> second = '150'
>>> print(first + second)
100150
```

Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so: the word, `Zebra`, comes before `apple`. A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison.

Practice with Operators & Expressions

Practice #1 - Use the Python IDLE Shell to calculate:

1. 6+4*10
2. (6+4)*10 (Compare this to the previous expression, and note that Python uses parentheses just like you would in normal math to determine order of operations!)
3. 23.0 to the 5th power
4. Positive root of the following equation: 34*x^2 + 68*x – 510
   Recall:
   a*x^2 + b*x + c
   x1 = (- b + sqrt (b*b - 4*a*c)) / (2*a)

Practice #2 - So now let us convert 645 minutes into hours. Use IDLE's Python Shell to type in the following:

```
>>> minutes = 645
>>> hours = minutes / 60
>>> hours
```

Oops! The output gives us `10.75` which is not what we expected. In Python 3, the division operator / always yields a floating point result. What we might have wanted to know was how many whole hours there are, and how many minutes remain. Python gives us two different flavors of the division operator. The second, called floor division uses the token //. Its result is always a whole number — and if it has to adjust the number it always moves it to the left on the number line. So 6 // 4 yields 1, but -6 // 4 might surprise you!

Practice #3 - Try this:

```
    >>> 7 / 4
    1.75
    >>> 7 // 4
    1
    >>> minutes = 645
    >>> hours = minutes // 60
    >>> hours
    10
```

Take care that you choose the correct flavor of the division operator. If you're working with expressions where you need floating point values, use the division operator that does the division accurately.

Evaluation Order

If you had an expression such as `2 + 3 * 4` , is the addition done first or the multiplication?

Recall from Algebra PEMDAS (parenthesis, exponents, multiplication, division, addition, subtraction).This tells us that the multiplication should be done first and that the multiplication operator has higher precedence than the addition operator.

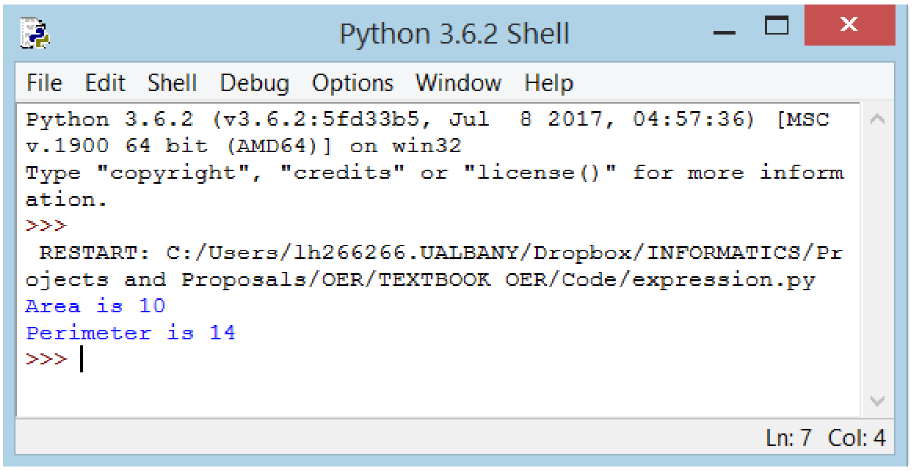
It is far better to use parentheses to group operators and operands appropriately in order to explicitly specify the precedence. This makes the program more readable. For example, `2 + (3 * 4)` is definitely easier to read than `2 + 3 * 4` which requires knowledge of the operator precedence's.

There is an additional advantage to using parentheses - it helps us to change the order of evaluation. For example, if you want addition to be evaluated before multiplication in an expression, then you can write something like `(2 + 3) * 4`.

Practice #4 - Use the Python editor to enter the following code (save as `expression.py`):

```
# given the length and width calculate the area and the
# perimeter of a rectangle
length = 5
width = 2
area = length * width
print('Area is', area)
print('Perimeter is', 2 * (length + width))
```

Output:



```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC
v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more inform
ation.
>>>
 RESTART: C:/Users/lh266266.UALBANY/Dropbox/INFORMATICS/Pr
ojects and Proposals/OER/TEXTBOOK OER/Code/expression.py
Area is 10
Perimeter is 14
>>> 
```

Let us examine how this program works.

Python Statement	Explanation

`length = 5`
`width = 2`
	The `length` and `width` of the rectangle are stored in variables by the same name. They are each assigned an integer value. We use these to calculate the `area` and `perimeter` of the rectangle with the help of expressions.

`area = length * width`
`print('Area is', area)`
	We store the result of (assign) the expression `length * width` in the variable `area` and then print it using the `print` function.

`print('Perimeter is', 2 * (length + width))`	In this `print` statement, we directly use the value of the expression `2 * (length + width)` in the `print` function. Also, notice how Python prints the output in a readable format even though we have not specified a space between `'Area is'` and the variable `area` (by adding a comma Python knows to separate the output as individual 'words', like you would in a sentence).

Input/Process/Output Pattern

Recall from our previous Unit the example input-process-output diagram for the algorithm we called `find_max`.

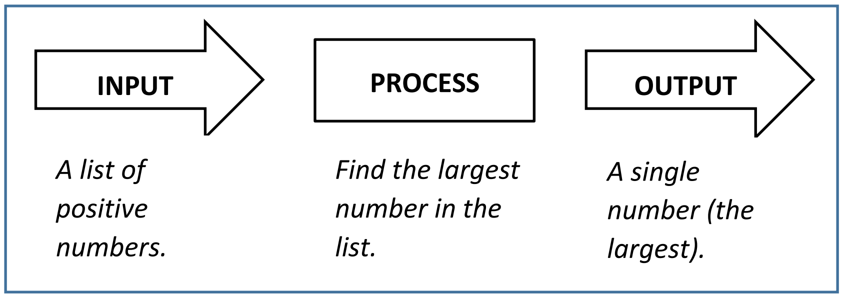

INPUT
A list of positive numbers.
PROCESS
Find the largest number in the list.
OUTPUT
A single number (the largest).

Figure 20: Input-Process-Output for Finding the Largest Number

We can represent any solution to a computational problem using this pattern of identifying the input (the data we are given) and then the general process that must be completed to get the desired output.

We have used Python statements invoking the print function to display a string of characters (i.e. the "message").

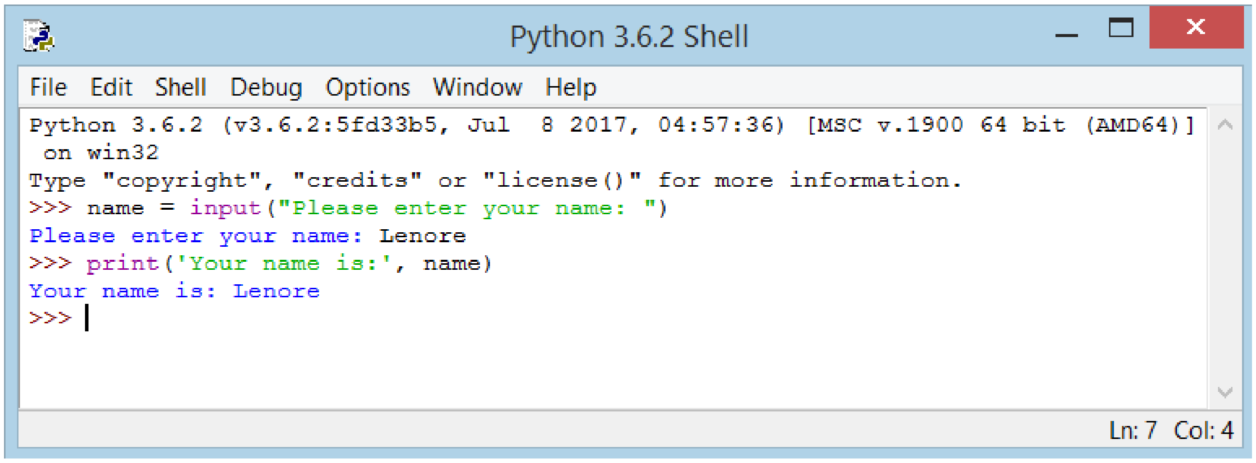
```
print("This is a simple Python program")
```

For starters, the built-in function `print()` will be used to print the output for our programs.

There is also a built-in function in Python for getting input from the user:

```
name = input("Please enter your name: ")
```

A sample run of this script in IDLE's Python Shell would pop up a window like this:



```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>> name = input("Please enter your name: ")
Please enter your name: Lenore
>>> print('Your name is:', name)
Your name is: Lenore
>>> 
```

Figure 21: Using the Built-in Function input()

The user of the program can enter the name and click OK (the *Enter* key), and when this happens the text that has been entered is returned from the `input` function, and in this case assigned to the variable `name`.

Even if you asked the user to enter their age, you would get back a string like "17". It would be your job, as the programmer, to convert that string into an integer or a float value.

Type Converter Functions

Here we'll look at three more built-in Python functions, `int()`, `float()` and `str()`, which will (attempt to) convert their arguments into data types int, float and str respectively. We call these type converter functions.

The int function can take a floating point number or a string, and turn it into an int. For floating point numbers, it discards the decimal portion of the number — a process we call truncation towards zero on the number line. For example:

```
>>> int(3.14)
3
>>> int(3.9999)            # This doesn't round to the closest int!
3
>>> int(3.0)
3
>>> int(-3.999)            # Note that the result is closer to zero
-3
>>> int(minutes / 60)
10
>>> int("2345")            # Parse a string to produce an int
2345
>>> int(17)                # It even works if arg is already an int
17
>>> int("23 bottles")
```

This last type conversion case doesn't look like a number — what do we expect?

```
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23 bottles'
```

The type converter `float()` can turn an integer, a float, or a syntactically legal string into a float:

```
>>> float(17)
17.0
>>> float("123.45")
123.45
```

The type converter `str()` turns its argument into a string:

```
>>> str(17)
'17'
>>> str(123.45)
'123.45'
```

If you are not sure what class a value falls into (i.e. unsure whether a value is an integer, a float or a string), Python has a built-in function called `type` which can tell you.

```
>>> type('hello')
<class 'str'>
>>> type(29)
<class 'int'>
>>> num = 89.32
>>> type(num)
<class 'float'>
```

Python’s Standard Library

As mentioned in Unit #1, Python has an extensive Standard Library which is a collection of built-in modules, each providing specific functionality beyond what is included in the “core” part of Python. A Python *module* is simply a file that contains Python code. The name of the file dictates the name of the module; for example, a file named `math.py` contains the functions available from the standard `math` module. We will explore this module (`math`) here.

Math Module and Math Functions

Python has a math module that provides most of the familiar mathematical functions. Before we can use the module, we have to import it:

```
>>> import math
```

This statement creates a module object named `math`. The following is a partial list of the functions which are provided by this module.

- `math.trunc(x)` : Returns the float value x truncated to an integer value.
- `math.sqrt(x)` : Returns the square root of x.
- `math.pow(x, y)` : Returns x raised to the power y.
- `math.degrees(x)` : Converts angle x from radians to degrees.
- `math.radians(x)` : Converts angle x from degrees to radians.

Many math operations depend on special constants also provided by the `math` module.

- `math.pi`: The mathematical constant π = 3.141592….
- `math.e`: The mathematical constant e = 2.718281….

Some examples of using the math module functions (note: these functions require the library name followed by a period followed by the function name):

```
import math
math.exp(5)                # returns 148.4131591025766
math.e**5                  # returns 148.4131591025765
math.sqrt(144)             # returns 12.0
math.pow(12.5, 2.8)        # returns 1178.5500657314767
math.pow(144, 0.5)         # returns 12.0
math.trunc(1.001)          # returns 1
math.trunc(1.999)          # returns 1
12*math.pi**2              #returns 18.4352528130723
```

Additional useful mathematical built-in functions besides `float()` and `int()` include:

- `abs(x)`: Returns the absolute value of the number x..
- `round( x [, n]  )`: returns x rounded to n digits from the decimal point (n is optional). If n is omitted it returns the nearest integer to its input.

More examples of math functions (no need to import the `math` module with these functions):

```
round(80.23456, 2)          # returns 80.23
round(100.000056, 3)        # returns 100.0
abs(-45)                    # returns 45
abs(100.12)                 # returns 100.12
```

More on Strings

We have seen how to print strings and how to get a string as input from a user. We also saw how to ‘add’ strings (concatenate) and ‘multiply’ strings.

```
>>> word1='fun'
>>> word2='times'
>>> word1+word2
'funtimes'
>>> word1*4
'funfunfunfun'
```

Processing of data includes the manipulation of strings (which is data) to produce something (information) which is meaningful. For example we might be presented with a file of thousands of passwords in use at a privately owned company and we would like to determine which of these passwords are secure and those that are not secure.

Recall that a string is a merely sequence of characters. To determine if an individual password was secure or not we might want to look at the length of the password and the individual characters in the password, looking for characters such as uppercase, numeric, special characters, etc.

Strings are actually a type of sequence; a sequence of individual characters. The indexing operator (Python uses square brackets to enclose the index) selects a single character substring from a string:

```
>>> pw = "abc123"
>>> char1 = pw[1]
>>> print(char1)
b
```

The expression `pw[1]` selects character number 1 from `pw`, and creates a new string containing just this one character. The variable `char1` refers to the result. When we display `char1`, we get the <u>second</u> character in the string pw, the letter “b”. Computer scientists always start counting from zero. The letter at subscript position zero of "`abc123`" is `a`. So at position [1] we have the letter `b`.

If we want to access the zero-eth letter of a string, we just place 0, or any expression that evaluates to 0, in between the brackets:

```
>>> pw = "abc123"
>>> char1 = pw[0]
>>> print(char1)
a
```

The expression in brackets is called an index. An index specifies a member of an ordered collection, in this case the collection of characters in the string. The index indicates which one you want, hence the name. It can be any integer expression.

Note that indexing returns a string — Python has no special type for a single character. It is just a string of length 1.

The string method `len()`, when applied to a string, returns the number of characters in a string:

```
>>> pw = "abc123"
>>> len(pw)
6
```

At some point, you may need to break a large string (i.g a paragraph) down into smaller chunks, or strings. This is the opposite of concatenation which merges or combines strings into one.

To do this, you use the `split()` method. What it does is split or breakup a string and add the data to a list of separate 'words' using a defined separator.

```
>>> sentence = "Python is an interpreted high-level programming
language for general-purpose programming."
>>> sentence.split()
['Python', 'is', 'an', 'interpreted', 'high-level', 'programming',
'language', 'for', 'general-purpose', 'programming.']
>>> len(sentence.split())
10
```

If no separator is defined when you call upon the function, whitespace will be used by default (as seen above). In simpler terms, the separator is a defined character that will be placed between each variable. For example:

```
>>> numbers = "122,35,09,97,56"
>>> numbers.split(",")
['122', '35', '09', '97', '56']
>>> len(numbers.split(","))
5
```

The string method `lower()` converts all lowercase characters in a string into uppercase characters and returns it.

```
>>> title="The Earth, My Butt, and Other Big Round Things"
>>> title.lower()
'the earth, my butt, and other big round things'
>>>
```

Similarly, The string method `upper()` converts all uppercase characters in a string into lowercase characters and returns it.

```
>>> title="Where the wild things are"
>>> title.upper()
'WHERE THE WILD THINGS ARE'
>>>
```

Unit Summary

TBD

- Explain the dependencies between hardware and software
- Describe the form and the function of computer programming languages
- Create, modify, and explain computer programs following the input/process/output pattern.
- Form valid Python identifiers and expressions.
- Write Python statements to output information to the screen, assign values to variables, and accept information from the keyboard.
- Read and write programs that process numerical data and the Python math module.
- Read and write programs that process textual data using built-in functions and methods.

From now on, we will assume that you have Python installed on your system. You should now be able to write, save and run Python programs at ease.

Now that you are a Python user, let us learn some more Python concepts.

We have seen how to use operators, operands and expressions - these are the basic building blocks of any program.

Next, we will see how to make use of these in our programs using statements.

We have seen how to use the three control flow statements - if , while and for along with their associated break and continue statements. These are some of the most commonly used parts of Python and hence, becoming comfortable with them is essential.

Practice Problems

1. What is the result of each of the following:
    a. >>> "Python"[1]
    b. >>> "Strings are sequences of characters."[5]
    c. >>> len("wonderful")
    d. >>> "Mystery"[:4]
    e. >>> "p" in "Pineapple"
    f. >>> "apple" in "Pineapple"
    g. >>> "pear" not in "Pineapple"

   h. >>> "apple" > "pineapple"
   i. >>> "pineapple" < "Peach"

2. Take the sentence: `All work and no play makes Jack a dull boy`. Store each word in a separate variable, then print out the sentence on one line using the `print` function.

3. Add parenthesis to the expression 6 * 1 - 2 to change its value from 4 to -6.

4. The formula for computing the final amount if one is earning compound interest is given on Wikipedia as this formula for compound interest:

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

Where,

- P = principal amount (initial investment)
- r = annual nominal interest rate (as a decimal)
- n = number of times the interest is compounded per year
- t = number of years

Write a Python program that assigns the principal amount of $10000 to variable `P`, assign to `n` the value 12, and assign to `r` the interest rate of 8%. Then have the program prompt the user for the number of years `t` that the money will be compounded for. Calculate and print the final amount after `t` years.

5. Evaluate the following numerical expressions on paper, then use the Python IDLE Shell to check your results:
   a. >>> 5 % 2
   b. >>> 9 % 5
   c. >>> 15 % 12
   d. >>> 12 % 15
   e. >>> 6 % 6
   f. >>> 0 % 7
   g. >>> 7 % 0

6. You look at the clock and it is exactly 2pm. You set an alarm to go off in 51 hours. At what time does the alarm go off? (Hint: you could count on your fingers, but this is not what we're after. If you are tempted to count on your fingers, change the 51 to 5100.)

   Write a Python program to solve the general version of the above problem. Ask the user for the time now (in hours), and ask for the number of hours to wait. Your program should output (print) what the time will be on the clock when the alarm goes off.

7. Write a program `find_hypot` which, given the length of two sides of a right-angled triangle, returns the length of the hypotenuse. (Hint: x ** 0.5 will return the square root.)

8. Practice using the Python IDLE Shell as a calculator:

a. Suppose the cover price of a book is $24.95, but bookstores get a 40% discount. Shipping costs $3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?

b. If I leave my house at 6:52 am and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast?

9. Enter the following statement, exactly, into the interactive Shell:

```
printt('What time is it?')
```

Is this a syntax error or a logic error?

10. Suppose that the math module of the Python Standard Library were imported. Write the Python statements to calculate the square root of four and print the answer.

11. What is the value of variables `num1` and `num2` after the following Python statements are executed?

```
num = 0
new = 5
num1 = num + new * 2
num2 = num + new * 2
```

12. What is wrong with the following statement that attempts to assign the value ten to variable x?

```
10 = x
```

13. Classify each of the following as either a legal or illegal Python identifier:

```
a. fred
b. if
c. 2x
d. -4
e. sum_total
f. sumTotal
g. sumtotal
h. While
i. x2
j. Private
k. public
```

l. `$16`
m. `xTwo`
n. `10%`
o. `a27834`

14. How is the value $\mathbf{2.45 \times 10^{-5}}$ expressed as a Python literal?

15. Given the following assignment:

```
x = 2
```

Indicate what each of the following Python statements would print.

```
a. print("x")
b. print('x')
c. print(x)
d. print("x + 1")
e. print('x' + 1)
f. print(x + 1)
```

16. Given the following assignments:

```
i1 = 2
i2 = 5
i3 = -3
d1 = 2.0
d2 = 5.0
d3 = -0.5
```

Evaluate each of the following Python expressions.

```
a. i1 + i2
b. i1 / i2
c. i1 // i2
d. i2 / i1
e. i1 * i3
f. d1 + d2
g. d1 / d2
h. d2 / d1
i. d3 * d1
j. d1 + i2
k. i1 / d2
```

```
l. d2 / i1
m. i2 / d1
n. i1/i2*d1
```

17. What is printed by the following statement:

```
#print(5/3)
```

18. Consider the following program which contains some errors. You may assume that the comments within the program accurately describe the program's intended behavior.

```
# Get two numbers from the user
n1 = float(input()) # 1
n2 = float(input()) # 2
# Compute sum of the two numbers
print(n1 + n2) # 3
# Compute average of the two numbers
print(n1+n2/2) # 4
# Assign some variables
d1 = d2 = 0 # 5
# Compute a quotient
print(n1/d1) # 6
# Compute a product
n1*n2 = d1 # 7
# Print result
print(d1) # 8
```

For each line listed in the comments, indicate whether or not an syntax error, run-time error, or semantic error is present. Not all lines contain an error.

19. What is printed by the following code fragment?

```
x1 = 2
x2 = 2
x1 += 1
x2 -= 1
print(x1)
print(x2)
```

20. Given the following definitions:

```
x = 3
```

```
y = 5
z = 7
```

evaluate the following Boolean expressions:

```
x == 3
x < y
x >= y
x <= y
x != y - 2
x < 10
x >= 0 and x < 10
x < 0 and x < 10
x >= 0 and x < 2
x < 0 or x < 10
x > 0 or x < 10
x < 0 or x > 10
```

21. Given the following definitions:

```
x =3
y = 3
b1 = true
b2 = false
x == 3
y < 3
```

evaluate the following Boolean expressions:

```
a. b2
b. not b1
c. not b2
d. b1 and b2
e. b1 or b2
f. x
g. y
h. x or y
i. not y
```

UNIT 3: Graphics: Designing and developing graphics programs.

Learning Objectives

- Explain what we mean by object-oriented programming.
- Apply the fundamental concepts of computer graphics in a computer program.
- Create objects in programs and use methods to perform graphical computations.
- Write simple interactive graphics programs using objects available in the graphics module.
- Read and write programs that define functions and use function calls and parameter passing in Python.

Object Oriented Programming[12]

Python is an object-oriented language. Every piece of data and even functions and types are objects. Most modern programming languages support object-oriented (OO) development to one degree or another. An object-oriented programming language allows the programmer to define, create, and manipulate objects. Objects bundle together data and functions. Like other variables, each Python object has a type, or class. The terms class and type are synonymous.

The term object-oriented is used to distinguish Python from earlier languages, classified as procedural languages, where types of data and the operations on them were not connected in the language. The functions we have used so far follow the older procedural programming syntax. In the newer paradigm of object-oriented programming, all data are in objects, and a core group of operations that can be done on some particular type of object are tightly bound to the object and called the object's methods.

Using Objects

An object is an instance of a class. We have been using objects since the beginning, but we have not taken advantage of all the capabilities that objects provide. Integers, floating-point numbers, strings, and functions are all objects in Python. With the exception of function objects, we have treated these objects as passive data. We can assign an integer value to a variable and then use that variable's value. We can add two floating-point numbers and concatenate two strings with the + operator. We can pass objects to functions and functions can return objects.

In object-oriented programming, rather than treating data as passive values and functions as active agents that manipulate data, we fuse data and functions together into software units called **objects**. A typical object consists of two parts: **data** and **methods**. An object's data consists of its instance variables. The term instance variable comes from the fact that the data is represented by a variable owned by an

[12] This section is based on Halterman, R. "Fundamentals of Python Programming." Southern Adventist University (2017).

object, and an object is an instance of a class. Other names for instance variables include attributes and fields. Methods are like functions, and they are known also as operations. The instance variables and methods of an object constitutes the object's members. The code that uses an object is called the object's client. We say that an object provides a service to its clients. The services provided by an object can be more elaborate that those provided by simple functions because objects make it easy to store persistent data in their instance variables.

In summary (Figure 22), objects are simply variables that are some defined data type (e.g. the variable "age" is an "integer" type). Objects have data associated with them (e.g. the variable "age" is assigned the value of 15 in the Python statement `age = 15`). To perform an operation (or procedure) on an object, we send the object a message. The set of messages an object responds to are called the methods of the object (e.g. the Python statement `print(age)`)

- Methods are like functions that live inside the object.
- Methods are invoked using dot-notation:
  `<object>.<method-name>(<param1>, <param2>, …)`

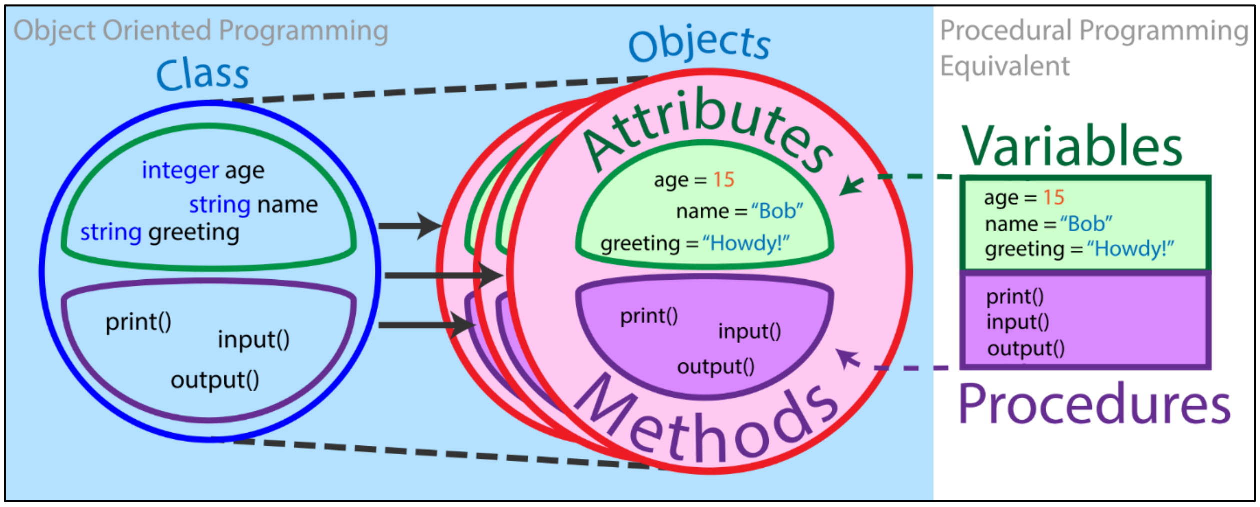


Figure 22: Object Oriented Programming (wikibooks.org/wiki/File:CPT-Object-Var-Proc.svg)

String Objects

We have been using string objects—instances of class `str`—for some time. Objects bundle data and functions together, and the data that comprise a string consist of the sequence of characters that make up the string. Strings are objects, and strings "know how" to produce an uppercase version of themselves. Try this in the Python IDLE Shell:

```
str = 'Hello!'
str.upper()
```

Here `upper` is a method associated with strings. This means `upper` is a function that is bound to the string before the dot. This function is bound both logically, and as we see in the new notation, also syntactically. One way to think about it is that each type of data knows operations (methods) that can be applied to it. The expression `str.upper()` calls the method upper that is bound to the string `str` and returns a new uppercase string result based on the variable `str`.

Strings are immutable, so no string method can change the original string, it can only return a new string. Confirm this by entering each line individually in the Python IDLE Shell to see the original `str` is unchanged:

```
str
str2 = s.upper()
str2
str
```

We are using the new object syntax:

```
    object.method( )
```

meaning that the method associated with the object's type is applied to the object. This is just a special syntax for a function call with an object.

Another string method is `lower`, analogous to `upper`, but producing a lowercase result.

Many methods also take additional parameters between the parentheses, using the more general syntax:

```
    object.method(parameters)
```

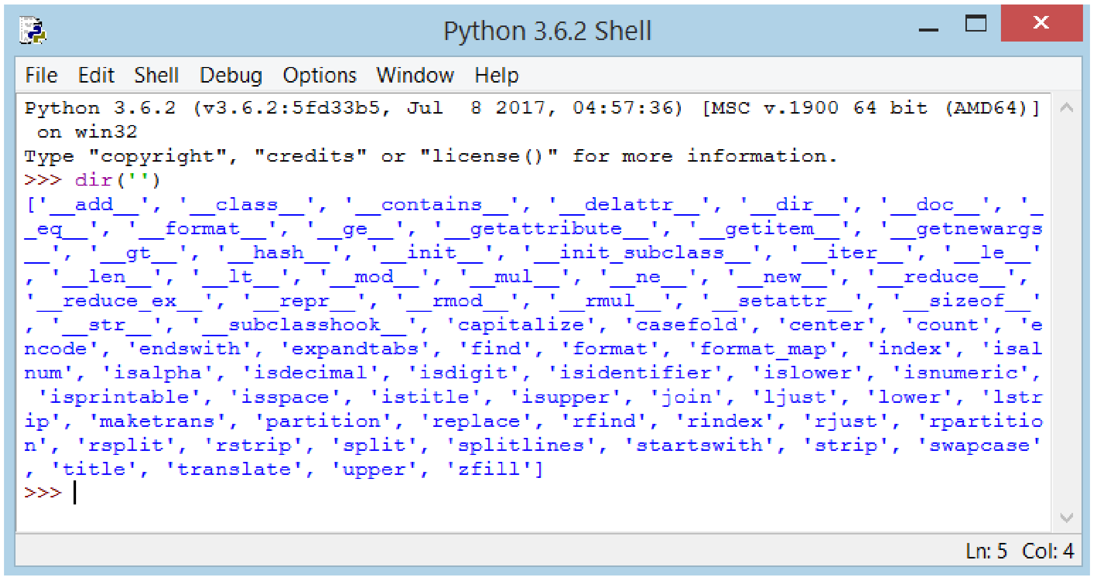
Technically the dot between the object and the method name is an operator, and operators have different levels of precedence. It is important to realize that this dot operator has the highest possible precedence. Read and see the difference parentheses make in the expressions (try this too!):

```
>>> 'hello ' + 'there'.upper()
'hello THERE'
>>> ('hello ' + 'there').upper()
'HELLO THERE'
```

Python lets you see all the methods that are bound to an object (and any object of its type) with the built-in function `dir`. To see all string methods, supply the `dir` function with any string. For example, try in the Python IDLE Shell:

```
dir('')
```

Many of the names in the list start and end with two underscores, like `__add__`. These are all associated with methods and pieces of data used internally by the Python interpreter. You can ignore them for now. The remaining entries in the list are all user-level methods for strings. You should see `lower` and `upper` among them. Some of the methods are much more commonly used than others.



```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>> dir('')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '_
_eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs
__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__'
, '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__'
, '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'e
ncode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isal
num', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstr
ip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartitio
n', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase'
, 'title', 'translate', 'upper', 'zfill']
>>> |
```

Figure 23: List of String Methods

Object notation:

```
object.method(parameters)
```

has been illustrated so far with just the object type `str`, but it applies to all types.

Simple Graphics Programming

Graphics make programming more fun for many people. To fully introduce graphics would involve many ideas that would be a distraction now. This section introduces a simplified graphics module developed by John Zelle for use with his Python Programming book.

In order to use this module and build graphical programs you will need a copy of `graphics.py`. You can download this from http://mcsp.wartburg.edu/zelle/python/graphics.py . You will want to place this file (`graphics.py`) in the same folder you use for all your Python programs (the files with the `.py` extension).

Note to Reader: You will just be a user of the `graphics.py` code, so you do not need to understand the inner workings! It uses all sorts of features of Python that are way beyond the scope of this book. There is no particular need to open `graphics.py` in the Python IDLE editor.

Once the module is downloaded (therefore 'installed'), we can import it just like a library that is built into Python like `math`:

```
import graphics
```

If typing this line in the Python IDLE Shell gives you no output, then you have installed the `graphics` module correctly! If there is an error message, then Python could not find the `graphics.py` file.

We will begin with a simple example program, `graphIntroSteps.py`[13], which you can download from here:

www.pas.rochester.edu/~rsarkis/csc161/_static/idle/examples/graphIntroSteps.py

Copy this `.py` file into the same folder that you copied the `graphics.py` file to and run it. Each time you press return, look at the screen and read the explanation for the next line(s).

Look around on your screen, and possibly underneath other windows: There should be a new window labeled "Graphics Window", created by the second line. Bring it to the top, and preferably drag it around to make it visible beside your Shell window. A `GraphWin` is a type of object from Zelle's graphics package that automatically displays a window when it is created. The assignment statement remembers the window object as win for future reference. (This will be our standard name for our graphics window object.) A small window, 200 by 200 pixels is created. A pixel is the smallest little square that can be displayed on your screen. Modern screens usually have more than 1000 pixels across the whole screen. A pixel is the basic unit of programmable color on a computer display or in a computer image.

The example program `graphIntro.py` starts with the same graphics code as `graphIntoSteps.py`, but without the need for pressing returns.

Added to this program is the ability to print a string value of `graphics` objects for debugging purposes. If some `graphics` object isn't visible because it is underneath something else or off the screen, temporarily adding this sort of output will help you debug your code. Let us examine each line of code in this graphics program.

Python Statement	Explanation
`from graphics import *`	Zelle's graphics are not a part of the standard Python distribution. For the Python interpreter to find Zelle's module, it must be imported.
`win = GraphWin()`	A `GraphWin` is a type of object that automatically displays a window when it is created. A small window, 200 by 200 pixels is created.
`pt = Point(100, 50)`	This creates a `Point` object and assigns it the name `pt`. A `Point` object, like each of the graphical objects that can be drawn on a `GraphWin`, has a method `draw`.
`pt.draw(win)`	Now you should see the `Point` if you look hard in the Graphics Window - it shows as a single, small, black pixel.
`cir = Circle(pt, 25)`	This creates a `Circle` object with center at the previously defined `pt` and with radius 25.
`cir.draw(win)`	Now you should see the `Circle` in the Graphics Window.
`cir.setOutline('red')`	Uses method `setOutline` to color the object's border.
`cir.setFill('blue')`	Uses method `setFill` to color the inside of the object.

[13] Written by: Harrington, Andrew N. "Hands-on Python Tutorial (Python 3.1 Version)." Loyola University Chicago. https://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/index.html#

`line = Line(pt, Point(150, 100))`	A `Line` object is constructed with two `Points` as parameters. In this case we use the previously named `Point`, `pt`, and specify another Point directly.
`line.draw(win)`	Technically the `Line` object is a segment between the the two points. You should now see the line in the Graphics Window.
`rect = Rectangle(Point(20, 10), pt)`	A rectangle is also specified by two points. The points must be diagonally opposite corners.
`rect.draw(win)`	The drawn `Rectangle` object is restricted to have horizontal and vertical sides.
`line.move(10, 40)`	The parameters to the move method are the amount to shift the x and y coordinates of the object. The line will visibly move in the Graphics Window.

`print('cir:', cir)`
`print('line:', line)`
`print('rect:', rect)`
	These three lines are used for debugging purposes. The coordinates for each object will be displayed in the Shell.

`input("Press return to end")`	This statement displays instructions (in the Shell) for the user.
`win.close()`	The Graphics Window is closed (it disappears).

Table 4: Python `graphIntro.py` Code Explained

Figure 24 shows the `Graphics` Window created by `graphIntroSteps.py` with the various graphics objects displayed in it. Next, in Figure 25, is the output that is displayed in the Python IDLE Shell. Once the user presses the Enter key ('return') the Graphics Window disappears.

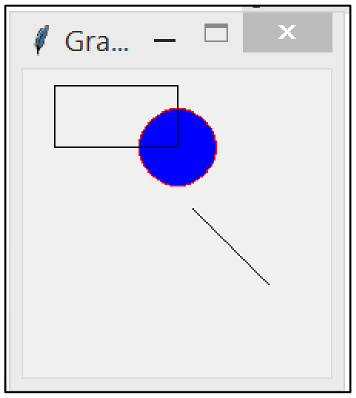
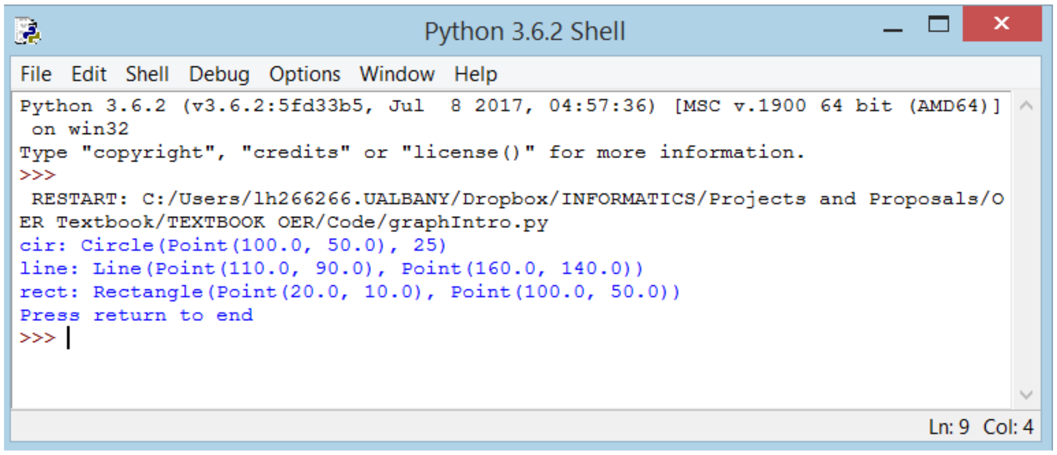


Figure 24: Example Graphics Window



```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:/Users/lh266266.UALBANY/Dropbox/INFORMATICS/Projects and Proposals/O
ER Textbook/TEXTBOOK OER/Code/graphIntro.py
cir: Circle(Point(100.0, 50.0), 25)
line: Line(Point(110.0, 90.0), Point(160.0, 140.0))
rect: Rectangle(Point(20.0, 10.0), Point(100.0, 50.0))
Press return to end
>>> 
```

Figure 25: Python Idle Shell Output

Graphics Windows: Coordinate Systems

Graphics windows have a Cartesian (x,y) coordinate system. The dimensions are initially measured in pixels. The first coordinate is the horizontal coordinate, measured from left to right, so in the default 200x200 window, 100 is about half way across the 200 pixel wide window. The second coordinate, for the vertical direction, increases going down from the top of the window by default, not up as you are likely to expect from geometry or algebra class. The coordinate 100 out of the total 200 vertically should be about 1/2 of the way down from the top. See figure 26.

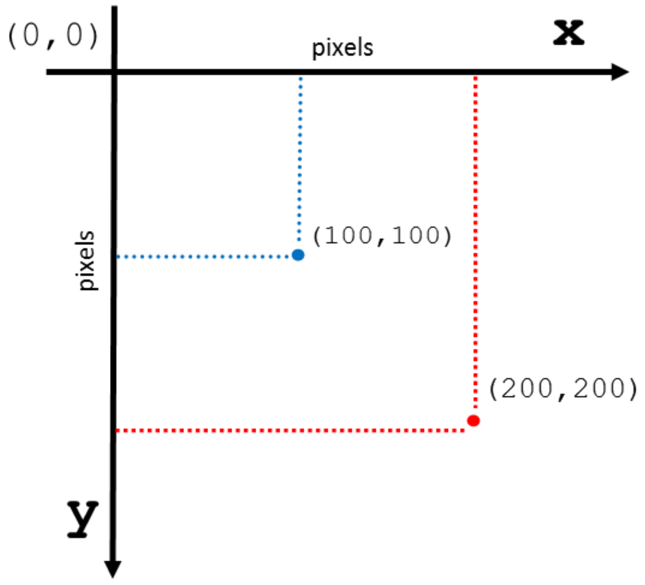


Figure 26: Default Graphics Coordinate System

Much of the graphics programming we will attempt will be based on designing simple GUIs (Graphical User Interfaces) based on some of the programs we have already written. Graphical user interfaces (GUIs) are human-computer interfaces allowing users to interact with an app. Rarely will you find a computer application today that doesn't have some sort of graphical interface to use it. A graphical interface consists of a window which is populated by "widgets", the basic building blocks of a GUI. Widgets are the pieces of a GUI that make it usable, e.g. buttons, icons, menu items, on-screen text boxes for typing in information, etc. Basically, anything the user can see in the window is a widget.

Placing and drawing objects in a window defined by pixels (creating images on the pixel level) can get very tedious. The `graphics.py` module provides us with the ability to set the coordinate system of the window using the `setCoords` method, which we will see in the following section.

GraphWin Objects[14]

A `GraphWin` object represents a window on the screen where graphical images may be drawn. A program may define any number of `GraphWin`s. A `GraphWin` understands the following methods. Recall

`GraphWin(title, width, height)`

Constructs a new graphics window for drawing on the screen.

The parameters are optional; the default title is "Graphics Window" and the default size is 200 x 200 pixels.

Example: `win = GraphWin("My App", 500, 180)`

The following figure shows the resulting graphics window which is 500 pixels wide and 180 pixels high with a title of "My App".

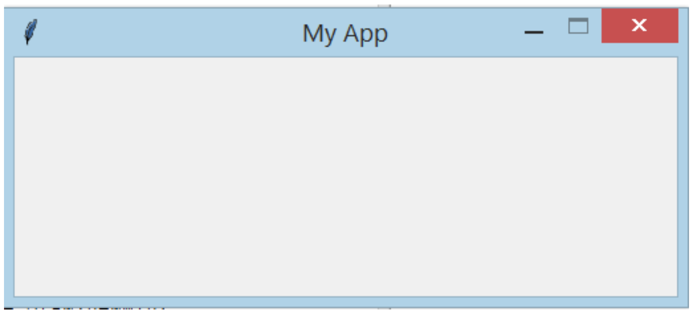


Figure 27: Default Graphics Window

`setBackground(color)` Sets the window background to the given color. The default background color depends on your system. See Section *Generating Colors* for information on specifying colors.

Example: `win.setBackground("white")`

`close()` Closes the on-screen window.

Example: `win.close()`

[14] Based on: Zelle, John M. "Graphics Module Reference." (2016).

`getMouse()` Pauses for the user to click a mouse in the window and returns where the mouse was clicked as a Point object.

Example: `clickPoint = win.getMouse()`

`setCoords(x1, y1, x2, y2)` Sets the coordinate system of the window. The lower-left corner is (x1; y1) and the upper-right corner is (x2; y2).

Example: `win.setCoords(0, 0, 4.0, 4.0)`

Example Program Using Coordinate Transformation

Suppose we wanted to create a simple game app where the user needed to match a pair of tiles with the goal of eventually pair-matching all the tiles in the grid. For example, the user might need to pair matching colored circles as seen here in Figure 28 by clicking on an empty pink square.

Drawing the objects of this GUI is certainly easier using coordinate transformation; e.g. defining the coordinate system of our GUI window to go from (0,0) in the lower left corner to (4,4) in the upper right corner).

Examine the code below (`matchColorsGame.py`).

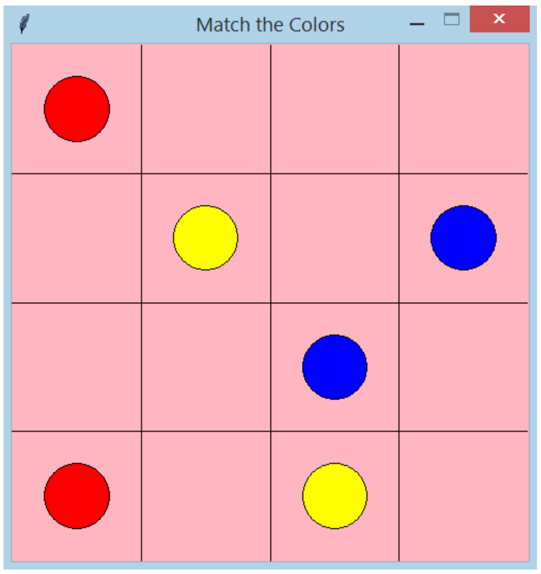


Figure 28: Match the Colors Game

```
from graphics import *

win = GraphWin("Match the Colors", 500, 500) #create a 500x500 window
# set coordinates to go from (0,0) in the lower left corner to
# (4,4) in the upper right.
win.setCoords(0.0, 0.0, 4.0, 4.0)
win.setBackground("LightPink")

# Draw the vertical lines
Line(Point(1,0), Point(1,4)).draw(win)
Line(Point(2,0), Point(2,4)).draw(win)
Line(Point(3,0), Point(3,4)).draw(win)

# Draw the horizontal lines
Line(Point(0,1), Point(4,1)).draw(win)
Line(Point(0,2), Point(4,2)).draw(win)
Line(Point(0,3), Point(4,3)).draw(win)
```

```
# Draw cirlcles in boxes
circle1 = Circle(Point(.5,.5), .25)
circle1.setFill('Red')
circle1.draw(win)
circle2 = Circle(Point(2.5,1.5), .25)
circle2.setFill('Blue')
circle2.draw(win)
circle3 = Circle(Point(1.5,2.5), .25)
circle3.setFill('Yellow')
circle3.draw(win)
circle4 = Circle(Point(3.5,2.5), .25)
circle4.setFill('Blue')
circle4.draw(win)
circle5 = Circle(Point(.5,3.5), .25)
circle5.setFill('Red')
circle5.draw(win)
circle6 = Circle(Point(2.5,.5), .25)
circle6.setFill('Yellow')
circle6.draw(win)

# wait for click and then quit
win.getMouse()
win.close()
```

Our program still defines the size of the window in pixels (the shaded area in the figure below), but the placement of objects in the window uses the coordinates defined using the `win.setCoords` method. See Figure 29.

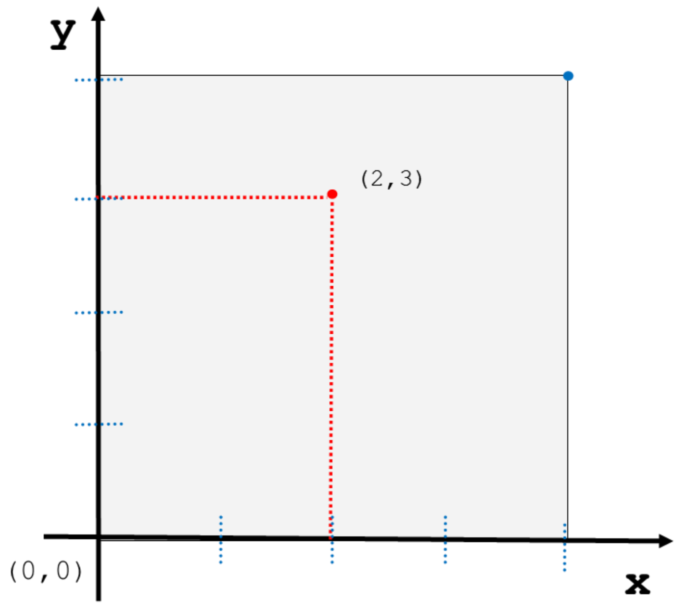


Figure 29: Transformed Graphics Coordinate System

Graphics Objects

The Graphics library provides the following classes of drawable objects: `Point, Line, Circle, Oval, Rectangle, Polygon`, and `Text`. All objects are initially created unfilled with a black outline. All graphics objects support the following generic set of methods:

`setFill(color)` Sets the interior of the object to the given color.

Example: `someObject.setFill("red")`

`setOutline(color)` Sets the outline of the object to the given color.

Example: `someObject.setOutline("yellow")`

`setWidth(pixels)` Sets the width of the outline of the object to the desired number of pixels. (Does not work for `Point`.)

Example: `someObject.setWidth(3)`

`draw(aGraphWin)` Draws the object into the given GraphWin and returns the drawn object.

Example: `someObject.draw(someGraphWin)`

`move(dx,dy)` Moves the object dx units in the x direction and dy units in the y direction. If the object is currently drawn, the image is adjusted to the new position.

Example: `someObject.move(10, 15.5)`

Point Methods

`Point(x,y)` Constructs a point having the given coordinates.

  Example: `aPoint = Point(3.5, 8)`

`getX()` Returns the x coordinate of a point.

  Example: `xValue = aPoint.getX()`

`getY()` Returns the y coordinate of a point.

  Example: `yValue = aPoint.getY()`

Line Methods

`Line(point1, point2)` Constructs a line segment from point1 to point2.

  Example: `aLine = Line(Point(1,3), Point(7,4))`

`setArrow(endString)` Sets the arrowhead status of a line. Arrows may be drawn at either the first point, the last point, or both. Possible values of `endString` are `"first"`, `"last"`, `"both"`, and `"none"`. The default setting is `"none"`.

  Example: `aLine.setArrow("both")`

`getCenter()` Returns a clone of the midpoint of the line segment.

  Example: `midPoint = aLine.getCenter()`

`getP1(), getP2()` Returns a clone of the corresponding endpoint of the segment.

  Example: `startPoint = aLine.getP1()`

Circle Methods

`Circle(centerPoint, radius)` Constructs a circle with the given center point and radius.

  Example: `aCircle = Circle(Point(3,4), 10.5)`

`getCenter()` Returns a clone of the center point of the circle.

  Example: `centerPoint = aCircle.getCenter()`

`getRadius`() Returns the radius of the circle.

  Example: `radius = aCircle.getRadius()`

`getP1(), getP2()` Returns a clone of the corresponding corner of the circle's bounding box. These are opposite corner points of a square that circumscribes the circle.

  Example: `cornerPoint = aCircle.getP1()`

Rectangle Methods

`Rectangle(point1, point2)` Constructs a rectangle having opposite corners at point1 and point2.

Example: `aRectangle = Rectangle(Point(1,3), Point(4,7))`

`getCenter()` Returns a clone of the center point of the rectangle.

Example: `centerPoint = aRectangle.getCenter()`

`getP1(), getP2()` Returns a clone of the corresponding point used to construct the rectangle.

Example: `cornerPoint = aRectangle.getP1()`

Oval Methods

`Oval(point1, point2)` Constructs an oval in the bounding box determined by point1 and point2.

Example: `anOval = Oval(Point(1,2), Point(3,4))`

`getCenter()` Returns a clone of the point at the center of the oval.

Example: `centerPoint = anOval.getCenter()`

`getP1(), getP2()` Returns a clone of the corresponding point used to construct the oval.

Example: `cornerPoint = anOval.getP1()`

Polygon Methods

`Polygon(point1, point2, point3, ...)` Constructs a polygon with the given points as vertices. Also accepts a single parameter that is a list of the vertices.

Example: `aPolygon = Polygon(Point(1,2), Point(3,4), Point(5,6))`

Example: `aPolygon = Polygon([Point(1,2), Point(3,4), Point(5,6)])`

`getPoints()` Returns a list containing clones of the points used to construct the polygon.

Example: `pointList = aPolygon.getPoints()`

Text Methods[15]

`Text(anchorPoint, textString)` Constructs a text object that displays textString centered at anchorPoint. The text is displayed horizontally.

Example: `message = Text(Point(3,4), "Hello!")`

[15] Based on: Zelle, John M. "Graphics Module Reference." (2016).

`setText(string)` Sets the text of the object to string.

Example: `message.setText("Goodbye!")`

`getText()` Returns the current string.

Example: `msgString = message.getText()`

`getAnchor()` Returns a clone of the anchor point.

Example: `centerPoint = message.getAnchor()`

`setFace(family)` Changes the font face to the given family. Possible values are `"helvetica"`, `"courier"`, `"times roman"`, and `"arial"`.

Example: `message.setFace("arial")`

`setSize(point)` Changes the font size to the given point size. Sizes from 5 to 36 points are legal.

Example: `message.setSize(18)`

`setStyle(style)` Changes font to the given style. Possible values are: `"normal"`, `"bold"`, `"italic"`, and `"bold italic"`.

Example: `message.setStyle("bold")`

`setTextColor(color)` Sets the color of the text to color.

Example: `message.setTextColor("pink")`

Entry Objects[16]

Objects of type `Entry` are displayed as text entry boxes that can be edited by the user of the program. `Entry` objects support the generic graphics `methods move(), draw(graphwin),` undraw(), setFill(color), and clone(). The Entry specific methods are given below.

`Entry(centerPoint, width)` Constructs an `Entry` having the given center point and width. The width is specified in number of characters of text that can be displayed.

Example: `inputBox = Entry(Point(3,4), 5)`

`getAnchor()` Returns a clone of the point where the entry box is centered.

Example: `centerPoint = inputBox.getAnchor()`

`getText()` Returns the string of text that is currently in the entry box.

Example: `inputStr = inputBox.getText()`

`setText(string)` Sets the text in the entry box to the given string.

Example: `inputBox.setText("32.0")`

[16] Based on: Zelle, John M. "Graphics Module Reference." (2016).

`setFace(family)` Changes the font face to the given family. Possible values are `"helvetica"`, `"courier"`, `"times roman"`, and `"arial"`.

    Example: `inputBox.setFace("courier")`

`setSize(point)` Changes the font size to the given point size. Sizes from 5 to 36 points are legal.

    Example: `inputBox.setSize(12)`

`setStyle(style)` Changes font to the given style. Possible values are: `"normal"`, `"bold"`, `"italic"`, and `"bold italic"`.

    Example: `inputBox.setStyle("italic")`

`setTextColor(color)` Sets the color of the text to color.

    Example: inputBox.setTextColor("green")

Displaying Images[17]

The Graphics library also provides minimal support for displaying and manipulating images in a `GraphWin`. Most platforms will support at least PPM and GIF images. Display is done with an Image object. Images support the generic methods `move(dx,dy)`, `draw(graphwin)`, `undraw()`, and `clone()`. Image-specific methods are given below.

`Image(anchorPoint, filename)` Constructs an image from contents of the given file, centered at the given anchor point. Can also be called with width and height parameters instead of filename. In this case, a blank (transparent) image is created of the given width and height (in pixels).

    Example: `flowerImage = Image(Point(100,100), "flower.gif")`

    Example: `blankImage = Image(320, 240)`

`getAnchor()` Returns a clone of the point where the image is centered.

    Example: `centerPoint = flowerImage.getAnchor()`

`getWidth()` Returns the width of the image.

    Example: `widthInPixels = flowerImage.getWidth()`

`getHeight()` Returns the height of the image.

    Example: `heightInPixels = flowerImage.getHeight()`

`getPixel(x, y)` Returns a list [red, green, blue] of the RGB values of the pixel at position (x,y). Each value is a number in the range 0-255 indicating the intensity of the corresponding RGB color. These numbers can be turned into a color string using the `color_rgb` function (see next section).

[17] Based on: Zelle, John M. "Graphics Module Reference." (2016).

Note that pixel position is relative to the image itself, not the window where the image may be drawn. The upper-left corner of the image is always pixel `(0,0)`.

Example: `red, green, blue = flowerImage.getPixel(32,18)`

`setPixel(x, y, color)` Sets the pixel at position (x,y) to the given color. Note: This is a slow operation.

Example: `flowerImage.setPixel(32, 18, "blue")`

`save(filename)` Saves the image to a file. The type of the resulting file (e.g., GIF or PPM) is determined by the extension on the filename.

Example: `flowerImage.save("mypic.ppm")`

Generating Colors[18]

Colors are indicated by strings. Most normal colors such as "red", "purple", "green", "cyan", etc. are available. Many colors come in various shades, such as "red1", "red2","red3", "red4", which are increasingly darker shades of red. For a full list, see the table below.

The graphics module also provides a function for mixing your own colors numerically. The function color rgb(red, green, blue) will return a string representing a color that is a mixture of the intensities of red, green and blue specified. These should be ints in the range 0-255. Thus `color rgb(255, 0, 0)` is a bright red, while `color rgb(130, 0, 130)` is a medium magenta.

Example: `aCircle.setFill(color rgb(130, 0, 130))`

'AliceBlue'	'firebrick'	'MistyRose'
'AntiqueWhite'	'ForestGreen'	'navy'
'aquamarine'	'gold'	'navy blue'
'azure'	'gray'	'OliveDrab'
'beige'	'gray1'	'orange'
'black'	'gray2'	'OrangeRed'
'BlanchedAlmond'	'gray3'	'orchid'
'blue'	'gray99'	'PaleGreen'
'BlueViolet'	'green'	'PaleTurquoise'
'brown'	'GreenYellow'	'PaleTurquoise1'
'CadetBlue'	'honeydew'	'PaleVioletRed'
'chartreuse'	'HotPink'	'PapayaWhip'
'chocolate'	'IndianRed'	'PeachPuff'
'coral'	'ivory'	'peru'
'CornflowerBlue'	'khaki'	'pink'
'cornsilk'	'lavender'	'plum'
'cyan'	'LavenderBlush'	'PowderBlue'

[18] Based on: Zelle, John M. "Graphics Module Reference." (2016).

'cyan1'	'LawnGreen'	'purple'
'cyan2'	'LemonChiffon'	'red'
'cyan3'	'light grey'	'RoyalBlue'
'cyan4'	'light slate gray'	'SaddleBrown'
'DarkBlue'	'LightBlue'	'salmon'
'DarkCyan'	'LightCoral'	'salmon1'
'DarkGoldenrod'	'LightCyan'	'salmon2'
'DarkGray'	'LightGoldenrod'	'salmon3'
'DarkGreen'	'LightGreen'	'salmon4'
'DarkGrey'	'LightPink'	'sandy brown'
'DarkKhaki'	'LightSalmon'	'SandyBrown'
'DarkMagenta'	'LightSeaGreen'	'SeaGreen'
'DarkOliveGreen'	'LightSkyBlue'	'seashell'
'DarkOrange'	'LightSlateBlue'	'sienna'
'DarkOrchid'	'LightSteelBlue'	'SkyBlue'
'DarkRed'	'LightYellow'	'SlateBlue'
'DarkSalmon'	'LimeGreen'	'SpringGreen'
'DarkSeaGreen'	'maroon'	'SpringGreen1'
'DarkSlateBlue'	'MediumAquamarine'	'SteelBlue'
'DarkSlateGray'	'MediumOrchid'	'tan'
'DarkTurquoise'	'MediumPurple'	'turquoise'
'DarkViolet'	'MediumSpringGreen'	'violet'
'DeepPink'	MediumTurquoise'	'VioletRed'
'DeepSkyBlue'	'MediumVioletRed'	'yellow'
'DimGray'	'MidnightBlue'	'YellowGreen'

Table 5: Graphics Library Color Names

Interactive Graphics

In a GUI environment, users interact with their applications by clicking on buttons, choosing items from menus, and typing information into on-screen text boxes. Event-driven programming draws interface elements (widgets) on the screen and then waits for the user to do something. An event is generated whenever a user moves the mouse, clicks the mouse, or types a key on the keyboard.

One limitation of the `graphics.py` module is that it is not robust if a graphics window is closed by clicking on the standard operating system close button on the title bar. If you close a graphics window that way, you are likely to get a Python error message. On the other hand, if your program creates a graphics window and then terminates abnormally due to some other error, the graphics window may be left orphaned. In this case the close button on the title bar is important: it is the easiest method to clean up and get rid of the window!

This lack of robustness is tied to the simplification designed into the `graphics` module. If the programmer wants user input, only one type can be specified at a time (either a mouse click in the graphics window via the getMouse method, or via the input keyboard Entry methods into the Shell window).

In graphIntro.py, a prompt to end the graphics program appeared in the Shell window, requiring you to pay attention to two windows.

```
input("Press return to end")
win.close()
```

In matchColorsGame.py, where all the action takes place in the graphics window, the only interaction is to click the mouse to close the graphics window. But as mentioned, clicking the standard operating system close button on the title bar will cause a Python Error. To close the graphics window without an error the user must click inside the window.

```
# wait for click and then quit
win.getMouse()
win.close()
```

Note to Reader: If you write a program with a bug, and the program gets an execution error while there is a GraphWin on the screen, a dead GraphWin lingers. The best way to clean things up is to make the Shell window be the current window and select from the menu Shell → Restart Shell.

Mouse Clicks

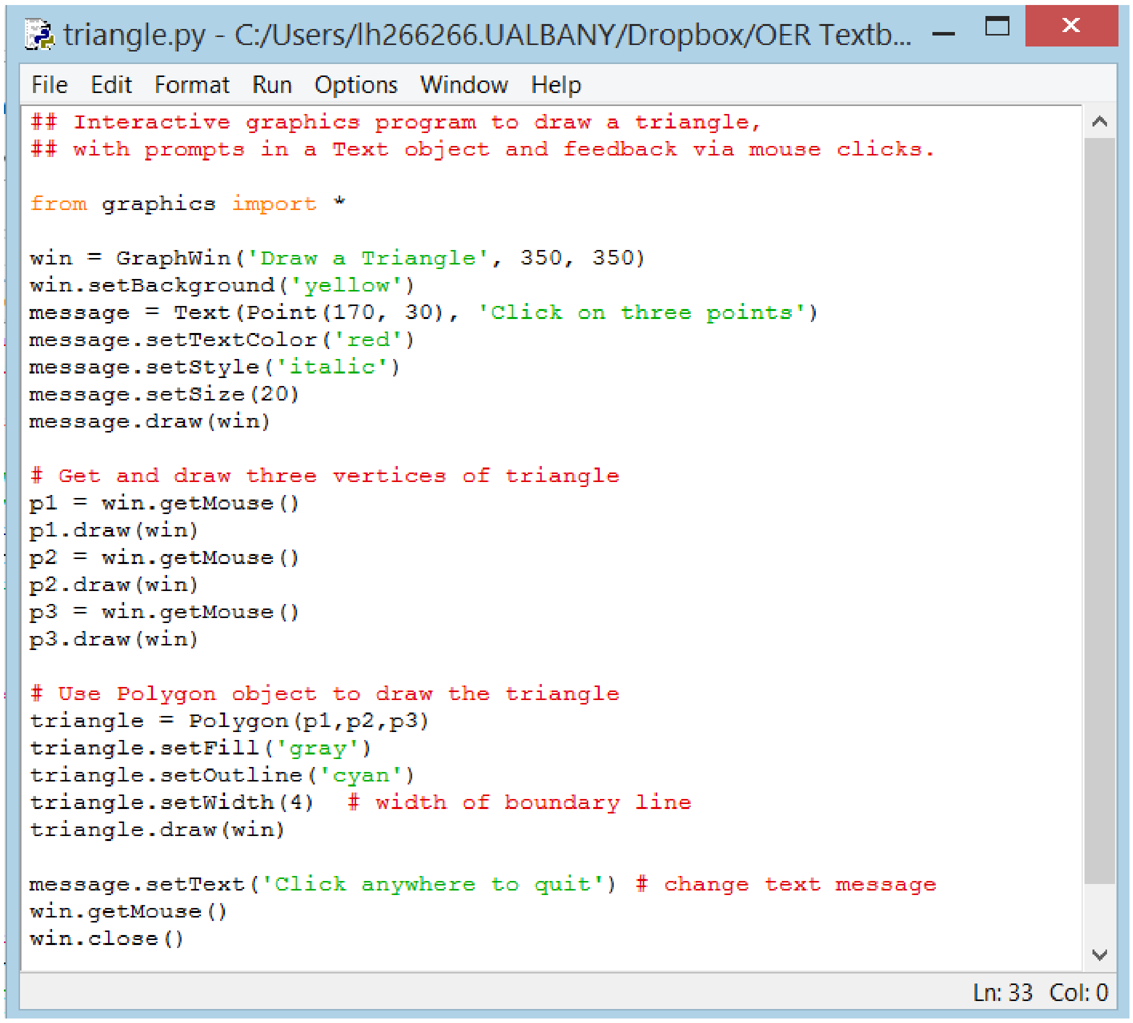
As we have seen in earlier examples, we can get graphical information from the user via the getMouse method of the GraphWin class. The code win.getMouse() waits for a mouse click. In the following code, the position of the mouse click is not important.

```
# wait for click and then quit
win.getMouse()
win.close()
```

When getMouse is invoked on a GraphWin, the program pauses and waits for the user to click the mouse somewhere in the window. The spot where the user clicked is returned as a Point object.

The next example, triangle.py[19], illustrates similar starting and ending code. In addition it explicitly interacts with the user. Rather than the code specifying literal coordinates for all graphical objects, the program remembers the places where the user clicks the mouse (stored in variables), and uses them as the vertices of a triangle.

[19] Based on code from: Harrington, Andrew N. "Hands-on Python Tutorial (Python 3.1 Version)." Loyola University Chicago. https://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/index.html#


triangle.py - C:/Users/lh266266.UALBANY/Dropbox/OER Textb...
File Edit Format Run Options Window Help
Ln: 33 Col: 0

```
## Interactive graphics program to draw a triangle,
## with prompts in a Text object and feedback via mouse clicks.

from graphics import *

win = GraphWin('Draw a Triangle', 350, 350)
win.setBackground('yellow')
message = Text(Point(170, 30), 'Click on three points')
message.setTextColor('red')
message.setStyle('italic')
message.setSize(20)
message.draw(win)

# Get and draw three vertices of triangle
p1 = win.getMouse()
p1.draw(win)
p2 = win.getMouse()
p2.draw(win)
p3 = win.getMouse()
p3.draw(win)

# Use Polygon object to draw the triangle
triangle = Polygon(p1,p2,p3)
triangle.setFill('gray')
triangle.setOutline('cyan')
triangle.setWidth(4)   # width of boundary line
triangle.draw(win)

message.setText('Click anywhere to quit') # change text message
win.getMouse()
win.close()
```

Figure 30: triangle.py

Let us examine the code in this graphics program.

Python Statement	Explanation

`from graphics import *`
`win = GraphWin('Draw a Triangle', 350, 350)`
`win.setBackground('yellow')`
	The standard starting lines (except for the specific values chosen for the width, height, and title of the window). The background color is a property of the whole graphics window that you can set.

`message = Text(Point(170, 30), 'Click on three points')`
`message.setTextColor('red')`
`message.setStyle('italic')`
`message.setSize(20)`
`message.draw(win)`
	A `Text` object is created. This is the prompt for user action. These lines illustrate most of the ways the appearance of a Text object may be modified, with results like in most word processors.

`p1 = win.getMouse()`
`p1.draw(win)`
`p2 = win.getMouse()`
	After the prompt, the program looks for a response. The `win.getMouse()` method (with no parameters), waits for you to click the mouse inside `win`. Then the `Point` where the mouse was clicked is returned. In this code three mouse

`p2.draw(win)`
`p3 = win.getMouse()`
`p3.draw(win)`
	clicks are waited for, remembered in variables `p1, p2, and p3`, and the points are drawn.

`triangle = Polygon(p1,p2,p3)`
`triangle.setFill('gray')`
`triangle.setOutline('cyan')`
`triangle.setWidth(4)   # width of boundary line`
`triangle.draw(win)`
	Next we see a very versatile type of graphical object, a Polygon, which may have any number of vertices specified in a list as its parameter. We see that the methods `setFill` and `setOutline` that we used earlier on a `Circle`, and the `setWidth` method we used for a `Line`, also apply to a Polygon, (and also to other graphics objects).

`message.setText('Click anywhere to quit') # change text message`
`win.getMouse()`
`win.close()`
	Besides changing the style of a Text object, the text itself may be changed as we see in the first line. The remaining code are standard ending lines.

Tip: Trying to figure out where to place objects in `GraphWin` can be time consuming and more than likely you will to need to adjust the positions of objects by trial and error until you get the positions you want. We can also print a descriptive string for each graphical type for debugging our graphics code. It only shows position, not other parts of the state of the object.

```
>>> pt = Point(30, 50)
>>> print(pt)
Point(30, 50)
>>> ln = Line(pt, Point(100, 150))
>>> print(ln)
```

Handling Textual Input

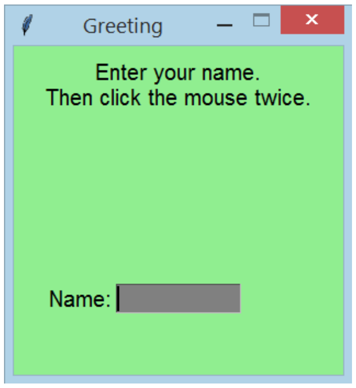
In the `triangle.py` example, all of the user input was provided through simple mouse clicks. The `graphics` module also provides a way for the user to enter text inside a textbox via the keyboard with the `Entry` type. This capability allows us to create primitive (but fully functional!) graphical user interfaces (GUIs). The `Entry` object is a partial replacement for the `input` function.

Let us look at a simple example, `greet.py`, which is presented below:

```
# Simple example with Entry objects (textboxes).
# Enter your name, click the mouse, and see greetings.

from graphics import *

win = GraphWin("Greeting", 300, 300)
win.setBackground('LightGreen')
instructions = Text(Point(150,35),
                 "Enter your name.\nThen click the mouse twice.")
instructions.draw(win) #display instructions

entry1 = Entry(Point(150, 230),10) #define the textbox
entry1.draw (win)   #draw the textbox

Text(Point (60, 230),'Name:').draw(win) # label for the textbox

win.getMouse()   # To know the user is finished with the text.

name = entry1.getText() #assign the text entered to the variable 'name'

greeting1 = 'Hello, ' + name + '!' # first greeting
Text(Point(110, 160), greeting1).draw(win)

greeting2 = 'Bonjour, ' + name + '!' # second greeting
Text(Point(160, 120), greeting2).draw(win)

win.getMouse() # To close the window
win.close()
```

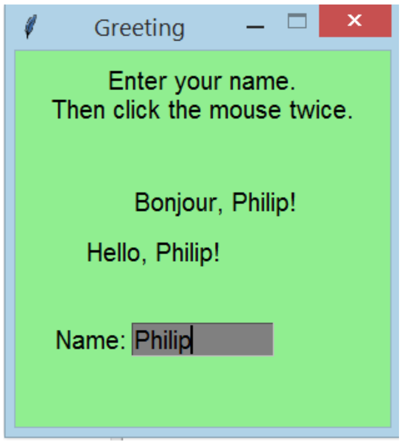
When we run this program the window displays (Figure 31), which allows the user to enter in a name.

The user then enters a name into the textbox in the window and is then instructed to 'click the mouse. Once the user clicks her mouse the first time, the window is updated to reflect the "greetings" as seen here in the second window (Figure 32). The user is also instructed to click her mouse again, which ends the program and closes the window.



```
Enter your name.
Then click the mouse twice.

Name:
```

Figure 32: greeting.py Opening Window



```
Enter your name.
Then click the mouse twice.

Bonjour, Philip!

Hello, Philip!

Name: Philip
```

Figure 31: greeting.py Updated Window

The last two graphics program, `trinagle.py` and `greeting.py`, placed and drew objects in a window defined by pixels which required needing to adjust the positions of objects by trial and error until we got the positions we wanted. This can become very tedious, very fast.

Let us look at how we can instead use coordinate transformation to more easily place and draw the objects in the `greeting.py` program. First we need to decide what coordinate system we want to apply for this app. One possibility is a three-by-three grid as seen here (Figure 33) by defining the coordinate system of our GUI window to go from (0,0) in the lower left corner to (3,3) in the upper right corner.

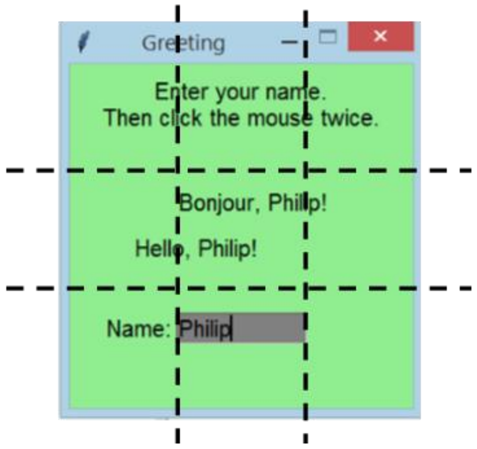
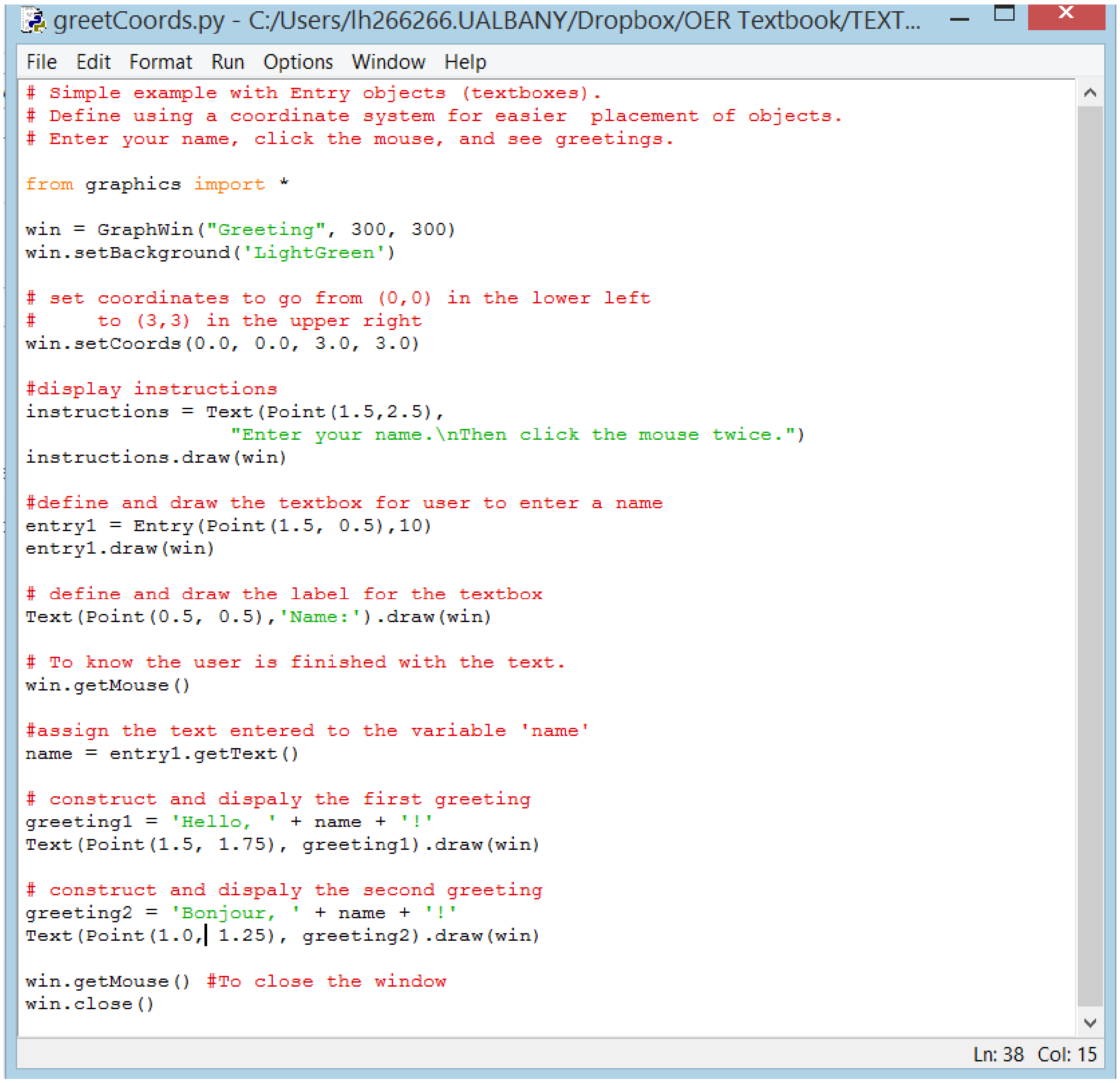


Figure 33: greeting.py with Coordinate Transformation

The code for the updated program using coordinate transformation made placing and displaying the objects easier (see figure 34). As we design and develop more complicated GUIs, defining our own coordinate system for each app is necessary.

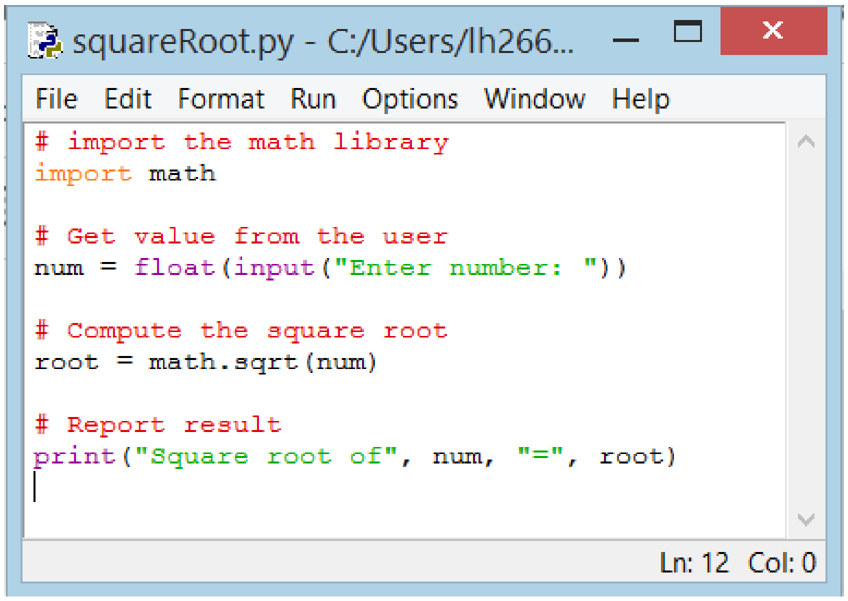

greetCoords.py - C:/Users/lh266266.UALBANY/Dropbox/OER Textbook/TEXT...
File Edit Format Run Options Window Help
Ln: 38 Col: 15

```
# Simple example with Entry objects (textboxes).
# Define using a coordinate system for easier  placement of objects.
# Enter your name, click the mouse, and see greetings.

from graphics import *

win = GraphWin("Greeting", 300, 300)
win.setBackground('LightGreen')

# set coordinates to go from (0,0) in the lower left
#     to (3,3) in the upper right
win.setCoords(0.0, 0.0, 3.0, 3.0)

#display instructions
instructions = Text(Point(1.5,2.5),
                 "Enter your name.\nThen click the mouse twice.")
instructions.draw(win)

#define and draw the textbox for user to enter a name
entry1 = Entry(Point(1.5, 0.5),10)
entry1.draw(win)

# define and draw the label for the textbox
Text(Point(0.5, 0.5),'Name:').draw(win)

# To know the user is finished with the text.
win.getMouse()

#assign the text entered to the variable 'name'
name = entry1.getText()

# construct and dispaly the first greeting
greeting1 = 'Hello, ' + name + '!'
Text(Point(1.5, 1.75), greeting1).draw(win)

# construct and dispaly the second greeting
greeting2 = 'Bonjour, ' + name + '!'
Text(Point(1.0, 1.25), greeting2).draw(win)

win.getMouse() #To close the window
win.close()
```

Figure 34:greetCoords.py

Functions

Functions are reusable pieces of programs. They allow you to give a name to a block of statements, allowing you to run that block using the specified name anywhere in your program and any number of times. This is known as <u>calling</u> the function. We have already used many Python built-in functions such as `input`, `print`, `int`, `str` and `float`. Additionally, the Python standard library includes many other functions useful for common programming tasks. Python has a function in its standard library named `sqrt`. The square root function accepts one numeric (integer or floating-point) value and produces a floating-point result; for example, $\sqrt{144} = 12$, so when presented with 144.0, `sqrt` returns the value 12.0. Try typing in and saving the following program:


squareRoot.py - C:/Users/lh266...
File Edit Format Run Options Window Help
Ln: 12 Col: 0

```
# import the math library
import math

# Get value from the user
num = float(input("Enter number: "))

# Compute the square root
root = math.sqrt(num)

# Report result
print("Square root of", num, "=", root)
```

Figure 35: square.py

In the Python statement `math.sqrt(num)`, the function `sqrt` is "called". The function `sqrt` requires a single "parameter" which in this example is the number (or value) that the user types in. The function accepts the parameter as the function's "argument" then "returns" the result which is assigned to the variable `root`.

The results of running this program is shown below:

```
Enter number: 144
Square root of 144.0 = 12.0
>>>
```

The function `sqrt` is like a black box; "callers" do not need to know the details of the code inside the function in order to use it (figure 36).

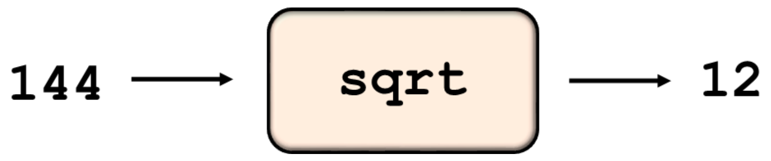


Figure 36: Conceptual view of the square root function

If the calling code attempts to pass a parameter to a function that is incompatible with the argument type expected by that function, the interpreter issues an error. Examine the results of trying this in the Python interactive Shell:

```
>>> import math
>>> math.sqrt(144)
12.0
>>> math.sqrt("144")
Traceback (most recent call last):
 File "<pyshell#2>", line 1, in <module>

    math.sqrt("144")

TypeError: must be real number, not str

>>>
```

The `sqrt` function can process only numbers: integers and floating-point numbers. Even though we know we could convert the string parameter `'16'` to the integer `16` (with the `int` function) or to the floating-point value `16:0` (with the `float` function), the `sqrt` function does not automatically do this for us.

Some functions take more than one parameter; for example, `print` can accept multiple parameters separated by commas.

From the caller's perspective a function has three important parts:

- **Name**. Every function has a name that identifies the code to be executed. Function names follow the same rules as variable names; a function name is another example of an identifier.
- **Parameters**. A function must be called with a certain number of parameters, and each parameter must be the correct type. Some functions like `print` permit callers to pass any number of arguments, but most functions, like `sqrt,` specify an exact number. If a caller attempts to call a function with too many or too few parameters, the interpreter will issue an error message and refuse to run the program (see examples below).

```
>>> import math
>>> math.sqrt(12.5)
3.5355339059327378
>>> math.sqrt()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    math.sqrt()
TypeError: sqrt() takes exactly one argument (0 given)
>>> math.sqrt(12,4.3)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    math.sqrt(12,4.3)
TypeError: sqrt() takes exactly one argument (2 given)
>>>
```

Similarly, if the parameters the caller passes are not compatible with the types specified for the function, the interpreter reports appropriate error messages.

- **Result type**. A function returns a value to its caller. Generally a function will compute a result and return the value of the result to the caller. The caller's use of this result must be compatible with the function's specified result type. A function's result type and its parameter types can be completely unrelated (see example below).

```
>>> import math
>>> type(math.sqrt(27))
<class 'float'>
>>>
```

Some functions do not accept any parameters; for example, the function to generate a pseudo-random floating-point number, `random,` requires no arguments (see below). The random function is part of the random module. The random function returns a floating-point value, but the caller does not pass the function any information to do its task.

```
>>> import random
>>> random.random()
0.6299660872157301
>>>
```

Defining Functions

As programs become longer and more complex, programmers must structure their programs in such a way as to effectively manage their complexity. We can write our own functions to divide our code into more manageable pieces. A modularized program is a program where each task within the program is in its own function. Besides their code organization aspects and ease of debugging, functions allow us to bundle functionality into reusable parts. Once a function is created, we can use (call) these functions in numerous places within a program. If the function's purpose is general enough and we write the function properly, we can reuse the function in other programs as well.

Functions are defined using the `def` keyword. After this keyword comes an identifier name for the function, followed by a pair of parentheses which may (or may not) enclose some names of variables, and by the final colon that ends the line. Function names should be descriptive of the task carried out by the function (and often includes a single verb indicating the single task the function is performing). Next follows the block of statements that are part of this function. An example, `function1.py,` will show that this is actually very simple:

```
# program to define a function and call the function
def say_hello():
    # block belonging to the function
    print('hello world')
    # End of function
say_hello() # call the function
say_hello() # call the function again
```

This program, `function1.py`, produces the following output in the interactive Shell:

```
>>>
 RESTART:
C:/Users/lh266266.UALBANY/Dropbox/OER
Textbook/TEXTBOOK OER/Code/function1.py
hello world
hello world
>>>
```

Let us examine how this works.

Python Statement	Explanation
`def say_hello():`	We define a function called `say_hello()`. This function takes no parameters and hence there are no variables declared in the parentheses. Parameters to functions are just input to the function so that we can pass in different values to it and get back corresponding results.
`print('hello world')`	This is the single line of code of the defined function. It is the only thing this function does.
`say_hello()` `say_hello()`	This is the call to the function. Notice that we can call the same function twice which means we do not have to write the same code again.

Function Parameters

A function can take parameters, which are values you supply to the function so that the function can do something using those values. These parameters are just like variables except that the values of these variables are defined when we call the function and are already assigned values when the function runs.

Parameters are specified within the pair of parentheses in the function definition, separated by commas. When we call the function, we supply the values in the same way. Note the terminology used -

the names given in the function definition are called **arguments** whereas the values you supply in the function call are called **parameters**.

Let us look at another program named `function2.py`.

```
# program to define a function with parameters and call the function
def print_max(a, b):
    if a > b:
        print(a, 'is maximum')
    elif a == b:
        print(a, 'is equal to', b)
    else:
        print(b, 'is maximum')

# directly pass literal values
print_max(3, 4)

#define 2 variables
num1 = 5
num2 = 7

# pass variables as arguments
print_max(num1, num2)
```

This program, `function2.py`, produces the following output in the interactive Shell:

```
>>>
 RESTART:
C:/Users/lh266266.UALBANY/Dropbox/OER
Textbook/TEXTBOOK OER/Code/function2.py
4 is maximum
7 is maximum
>>>
```

Let us examine how the program `function2.py` works.

Python Statement	Explanation

`def print_max(a, b):`
`    if a > b:`
`        print(a, 'is maximum')`
`    elif a == b:`
	We define a function called `print_max()` that uses two arguments called `a` and `b`.

`        print(a, 'is equal to', b)`
`    else:`
`        print(b, 'is maximum')`
	We find out the greater number using a simple `if..else` statement and then print the bigger number.

`print_max(3, 4)`	The first time we call the function `print_max`, we directly supply the numbers as parameters.

`num1 = 5`
`num2 = 7`
`print_max(num1, num2)`
	In the second case, we call the function with variables as parameters.
	The statement `print_max(num1, num2)` causes the value of argument `num1` to be assigned to parameter `a` and the value of argument `num2` to be assigned to parameter `b`. The `print_max` function works the same way in both cases.

The `return` Statement

The `return` statement is used to return from a function i.e. break out of the function. We can optionally return a value from the function as well.

Let us look at another program named `function3.py` that uses the return statement.

```
# program to define a function with parameters and a return
statement
def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return 'The numbers are equal'
    else:
        return y

# test case 1: x < y
print(maximum(2, 3))

# test case 2: x = y
print(maximum(2, 2))

# test case 3: x > y
print(maximum(2, 1))
```

This program, `function3.py`, produces the following output in the interactive Shell:

```
>>>
 RESTART:
C:/Users/lh266266.UALBANY/Dropbox/OER
Textbook/TEXTBOOK OER/Code/function3.py
3
The numbers are equal
2
>>>
```

Let us examine how the program `function3.py` works.

Python Statement	Explanation

`def maximum(x, y):`
`    if x > y:`
`        return x`
`    elif x == y:`
`        return 'The numbers are equal'`
`    else:`
`        return y`
	We define a function called `maximum()` that uses two arguments called `x` and `y` .
	It uses a simple `if..else` statement to find the greater value and then returns that value.

`# test case 1: x < y`
`print(maximum(2, 3))`
	The first time we call the function `maximum`, we are testing for `x` less than `y`.

`# test case 2: x = y`
`print(maximum(2, 2))`
	The second time we call the function `maximum`, we are testing for `x` equal to `y`.

`# test case 3: x > y`
`print(maximum(2, 1))`
	The third time we call the function `maximum`, we are testing for `x` greater than `y`.

Note that a `return` statement without a value is equivalent to `return None`. `None` is a special type in Python that represents nothingness. For example, it is used to indicate that a variable has no value if it has a value of `None` .

Every function implicitly contains a return `None` statement at the end unless you have written your own return statement.

Practice

Let us examine the following program which calculates the distance between two points. We use variables `x1` and `y1` to represent the (x,y) position of the first point and variables `x2` and `y2` to represent the (x,y) position of the second point. The formula for finding the distance between two points requires the `math` library since we need to use the square root function.

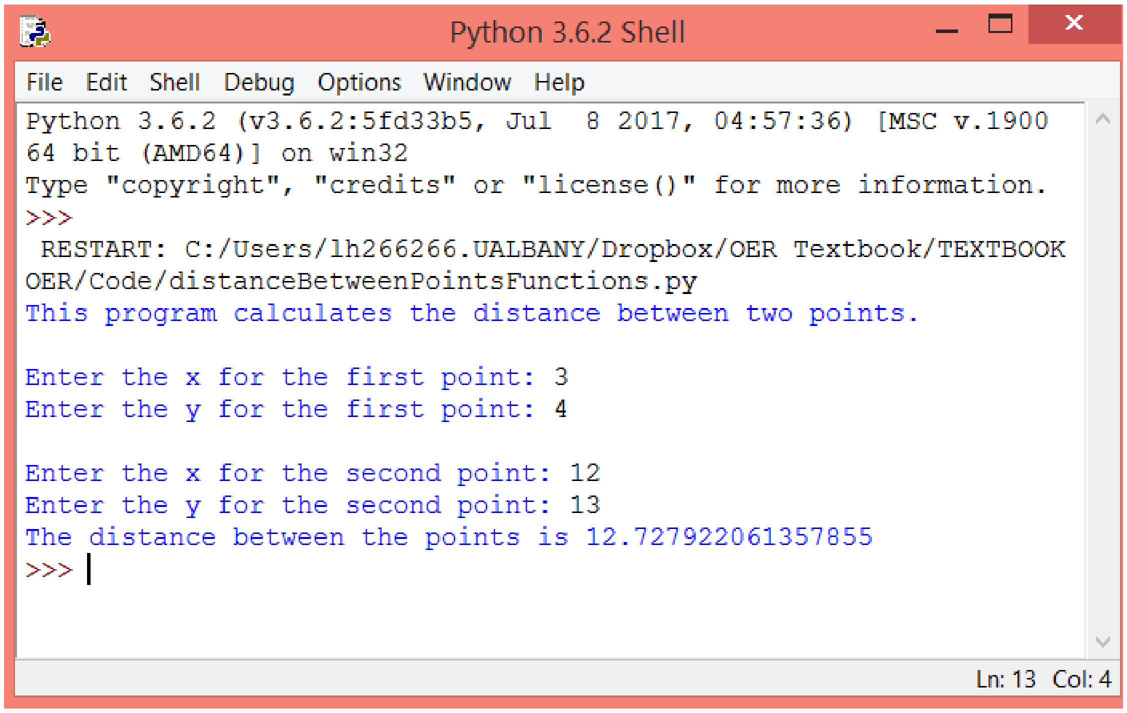
```
# Calculates the distance between two points

import math

print("This program calculates the distance between two points.")
print() #print a blank line

x1 = float(input("Enter the x for the first point: "))
y1 = float(input("Enter the y for the first point: "))
print()
x2 = float(input("Enter the x for the second point: "))
y2 = float(input("Enter the y for the second point: "))

distance = math.sqrt((x2-x1)**2 + (y2-y1)**2)

print()
print("The distance between the points is", distance)
```

Now, let us examine the same program rewritten to include a function to determine the distance between any two points (x1,y1) and (x2, y2).

```
# Calculates the distance between two points using a function

import math

#define the function
def distance(point1x,point1y, point2x,point2y):
    dist = math.sqrt((point2x-point1x)**2 + (point2y-point1y)**2)
    return dist

# main part of the program
print("This program calculates the distance between two points.")
print()

x1 = float(input("Enter the x for the first point: "))
y1 = float(input("Enter the y for the first point: "))
print()
x2 = float(input("Enter the x for the second point: "))
y2 = float(input("Enter the y for the second point: "))

print("The distance between the points is",
distance(x1,y1,x2,y2))
```

Notice that we define the function before we write the "main" part of the program. This is a standard practice when modularizing a program using functions.

The results of executing this program with point 1 being `(3,4)` and point 2 `(12,13)` looks like the following in the Python shell:



```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900
64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:/Users/lh266266.UALBANY/Dropbox/OER Textbook/TEXTBOOK
OER/Code/distanceBetweenPointsFunctions.py
This program calculates the distance between two points.

Enter the x for the first point: 3
Enter the y for the first point: 4

Enter the x for the second point: 12
Enter the y for the second point: 13
The distance between the points is 12.727922061357855
>>> 
```

Recall the `triangle.py` program we worked with earlier in this unit. Let us expand on this program to include additional textual output. The input for our revised `triangle.py` program, named `triangleGUIfunctions.py`[20], remains the same; the user clicks on three different points in the Graphics window. The output for our revised program still includes the graphic representing the triangle and, additionally, the perimeter of the graphic triangle as textual output. We use the *distance-between-two-points* function we just wrote!

[20] Based on Zelle, John M. *Python programming: an introduction to computer science*. 3rd ed., Franklin, Beedle & Associates, Inc., 2017, pp. 189-190.

```
import math
from graphics import *

#define the distance function with two arguments
def distance(p1, p2):
    dist = math.sqrt((p2.getX() - p1.getX()) **2 + (p2.getY() -
p1.getY())**2)
    return dist

#main part of the program
win = GraphWin("Draw a Triangle",500,500)
win.setCoords(0.0, 0.0, 10.0, 10.0)
message1 = Text(Point(5, 1), "Click on three points")
message1.draw(win)

# Get and draw three vertices of triangle
p1 = win.getMouse()
p1.draw(win)
p2 = win.getMouse()
p2.draw(win)
p3 = win.getMouse()
p3.draw(win)

# Use Polygon object to draw the triangle
triangle = Polygon(p1,p2,p3)
triangle.setFill("yellow")
triangle.setOutline("cyan")
triangle.draw(win)

# Calculate the perimeter of the triangle
# Call the distance function 3 times to find the length of each side of
the triangle
d1 = distance(p1,p2)
d2 = distance(p2,p3)
d3 = distance(p3,p1)
msg = "perimeter:" + str(d1+d2+d3)
message1.setText(msg)

# Wait for another click to exit
win.getMouse()
```

Let us examine the code of `triangleGUIfunctions.py` more closely and see how it works:

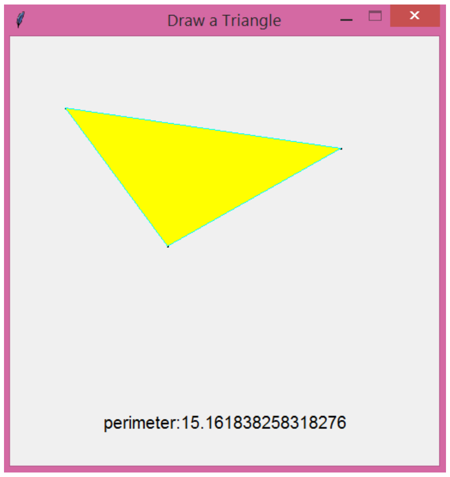
Python Statement	Explanation

`def distance(p1, p2):`
`    dist = math.sqrt((p2.getX() -`
`p1.getX()) **2 + (p2.getY() -`
`p1.getY())**2)`
`    return dist`
	We use our `distance` function modified to use only <u>two</u> arguments called `p1` and `p2`, representing the two vertices of the triangle we want to find the length of.
	We use the Point methods `getX()` and `getY()` to extract the individual `x` and `y` values for each point.

`win = GraphWin("Draw a Triangle",500,500)`
`win.setCoords(0.0, 0.0, 10.0, 10.0)`
`message1 = Text(Point(5, 1), "Click on`
`three points")`
`message1.draw(win)`
 	
`# Get and draw three vertices of triangle`
`p1 = win.getMouse()`
`p1.draw(win)`
`p2 = win.getMouse()`
`p2.draw(win)`
`p3 = win.getMouse()`
`p3.draw(win)`
 	
`# Use Polygon object to draw the triangle`
`triangle = Polygon(p1,p2,p3)`
`triangle.setFill("yellow")`
`triangle.setOutline("cyan")`
`triangle.draw(win)`
	This is the same code we used in `triangle.py` to accept the three points and then draw the triangle.

`# Calculate the perimeter of the triangle`
`# Call the distance function 3 times to`
`find the length of each side of the`
`triangle`
`d1 = distance(p1,p2)`
`d2 = distance(p2,p3)`
`d3 = distance(p3,p1)`
`msg = "perimeter:" + str(d1+d2+d3)`
`message1.setText(msg)`
 	
`# Wait for another click to exit`
`win.getMouse()`
	To determine the length of each side of the triangle we call the `distance` function three times. Adding the lengths of each side of the triangle gives us the perimeter.
	We print this value (`msg`) in a text a text box in the Graphics window.

The output from running `triangleGUIfunctions.py` is seen here in figure 37.



```
perimeter:15.161838258318276
```

Figure 37: triangleGUIfunctions.py Graphics window output

Unit Summary

TBD

- Apply the fundamental concepts of computer graphics in a computer program.
- Create objects in programs and use methods to perform graphical computations.
- Write simple interactive graphics programs using objects available in the graphics module.
- Read and write programs that define functions and use function calls and parameter passing in Python.

We have seen many aspects of functions but note that we still haven't covered all aspects of them. However, we have already covered much of what you will see and use in a beginning programming course.

Practice Problems

1. Make a program scene.py creating a scene with the graphics methods. You are likely to need to adjust the positions of objects by trial and error until you get the positions you want. Make sure you have graphics.py in the same directory as your program.
2. Elaborate the scene program above so it becomes changeScene.py, and changes one or more times when you click the mouse (and use win.getMouse()). You may use the position of the mouse click to affect the result, or it may just indicate you are ready to go on to the next view.

3. Is the following a legal Python program?

```
def proc(x):
    return x + 2
def proc(n):
    return 2*n + 1
def main():
    x = proc(5)
main()
```

4. Is the following a legal Python program?

```
def proc(x):
    return x + 2
def main():
    x = proc(5)
    y = proc(4)
main()
```

5. Is the following a legal Python program?

```
def proc(x):
    return 2*x
def main():
    print(proc(5, 4))
main()
```

6. The programmer was expecting the following program to print 200. What does it print instead? Why does it print what it does?

```
def proc(x):
    x = 2*x*x
def main():
    num = 10
    proc(num)
```

7. Complete the following distance function that computes the distance between two geometric points (x1;y1) and (x2;y2):

```
def distance(x1, y1, x2, y2):
...
```

Test it with several points to convince yourself that is correct.

8. A number, a, is a power of b if it is divisible by b and a/b is a power of b. Write a function called `ispower` that takes parameters a and b and returns `True` if a is a power of b. Note: you will have to think about the base case.

UNIT 4: Control Structures: Making Decisions and Looping in computing. Data and Information Processing in Python.

Learning Objectives

- Read and write programs using the Python IF and IF/ELIF/ELSE statements to implement a simple decision structures.
- Write simple exception handling code to catch simple Python run-time errors.
- Read and write programs using the Python FOR and WHILE statements to implement a simple loop structures.
- Construct and implement algorithms that use decision and loop structures.
- Apply basic file processing concepts and techniques for reading and writing text files in Python.

Boolean Expressions

Arithmetic expressions evaluate to numeric values; a Boolean expression may have only one of two possible values: **false** or **true**. While on the surface Boolean expressions may appear very limited compared to numeric expressions, they are essential for building more interesting and useful programs.

The simplest Boolean expressions in Python are `True` and `False`. In the Python interactive shell we see:

```
>>> True
True
>>> False
False
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

We see that `bool` is the name of the class representing Python's Boolean expressions. The following code (`booleanVars.py`) is a simple program that shows how Boolean variables can be used.

```
# Assign some Boolean variables
a = True
b = False
print('a =', a, ' b =', b)
# Reassign a
a = False
print('a =', a, ' b =', b)
```

The results of running this program is shown below:

```
a = True  b = False
a = False  b = False
>>>
```

A Boolean variable is also a Boolean expression as we saw in Unit 2. An expression comparing numeric expressions for equality or inequality is also a Boolean expression. The simplest kinds of Boolean expressions use relational operators (or comparison operators) to compare two expressions. Table 6 lists the relational operators available in Python.

Expression	**Meaning**
`x == y`	True if x *equals* y (mathematical equality, not assignment); otherwise, false
`x < y`	True if x *less than* y; otherwise, false
`x <= y`	True if x *less than and equal to y*; otherwise, false
`x > y`	True if x *greater than* y; otherwise, false
`x >= y`	True if x *greater than and equal to y*; otherwise, false
`x != y`	True if x *not equal to* y; otherwise, false

Table 6: Python Relationa/Comparisonl Operators

In the Python interactive shell we see some examples of Boolean expressions:

```
>>> 10 < 20
True
>>> 10 >= 20
False
>>> x = 19
>>> y = 29
>>> x < 100
True
>>> x != y
True
```

An expression like `10 < 20` is legal but of little use, since `10 < 20` is always true; the expression `True` is equivalent, simpler, and less likely to confuse human readers. Since variables can change their values during a program's execution (and often do!), Boolean expressions are most useful when their truth values depend on the values of one or more variables.

The relational/comparison operators are binary operators and are all left associative. They all have a lower precedence than any of the arithmetic operators; therefore, Python evaluates the expression

```
x + 2 < y / 10
```

as if parentheses were placed as so:

```
(x + 2) < (y / 10)
```

Logical operators

There are three logical operators: `and`, `or`, and `not`. The semantics (meaning) of these operators is similar to their meaning in English. For example, `x > 0 and x < 10` is true only if `x` is greater than 0 and less than 10.

`n%2 == 0 or n%3 == 0` is true if either or both of the conditions is true, that is, if the number is divisible by 2 or 3.

Finally, the `not` operator negates a Boolean expression, so `not (x > y)` is true if `x > y` is false, that is, if `x` is less than or equal to `y`.

Strictly speaking, the operands of the logical operators should be Boolean expressions, but Python is not very strict. Any nonzero number is interpreted as `True`:

```
>>> 42 and True
True
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it (unless you know what you are doing).

Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the **if** statement:

```
if x > 0:
    print('x is positive')
```

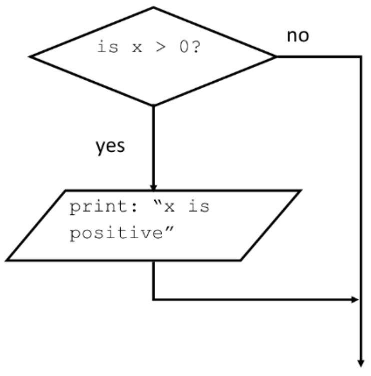


Figure 38: Simple IF Conditional Flowchart

The Boolean expression after `if` is called the condition. If it is true, the indented statement runs. If not, nothing happens. See the corresponding flowchart in figure 38.

`if` statements have the same structure as function definitions: a header followed by an indented body. Statements like this are called compound statements.

There is no limit on the number of statements that can appear in the body, but there has to be at least one.

A second form of the `if` statement is "alternative execution" (**`if-else`**), in which there are two possibilities and the condition determines which one runs. The syntax looks like this:

```
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

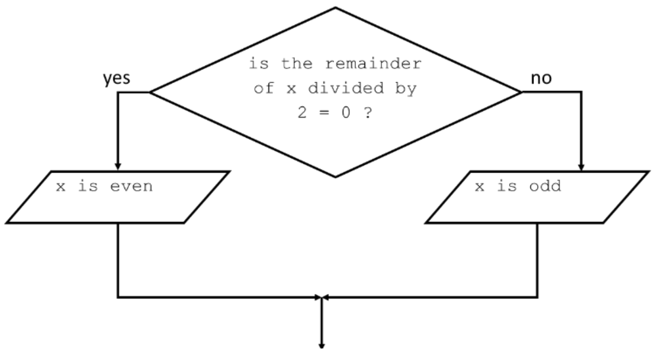


Figure 39: Alternative Execution Conditional Flowchart

Modulus or remainder operator, '`%`', returns the remainder of two numbers. So in this example, if the remainder when `x` is divided by 2 is 0, then we know that `x` is even, and the program displays an appropriate message. If the condition is false, the second set of statements runs. Since the condition must be true or false, exactly one of the alternatives will run. The alternatives are called branches, because they are branches in the flow of execution. See the corresponding flowchart in figure 39.

The `if` statement is used to check a condition: *if* the condition is true, we run a block of statements (called the *if-block*), *else* we process another block of statements (called the *else-block*). The *else* clause is optional. In our earlier examples we only had one statement in a block. Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional (**`if-elif-else`**). The following program, `ifElse.py`, shows us multiple statements in each block.

```
number = 23
guess = int(input('Enter an integer : '))

if guess == number:
    # New block starts here
    print('Congratulations, you guessed it.')
    print('(but you do not win any prizes!)')
    # New block ends here
elif guess < number:
    # Another block
    print('No, it is a little higher than that')
    # You can do whatever you want in a block ...
else:
    print('No, it is a little lower than that')
    # you must have guessed > number to reach here

print('Done')
# This last statement is always executed,
# after the if statement is executed.
```

In this program, we take guesses from the user and check if it is the number that we have. We set the variable `number` to any integer we want, say 23. Then, we take the user's guess using the `input()` function. We then convert this string to an integer using `int` and then store it in the variable `guess`.

Next, we compare the guess of the user with the number we have chosen. If they are equal, we print a success message. Notice that we use indentation levels to tell Python which statements belong to which block. Then, we check if the guess is less than the number, and if so, we inform the user that they must guess a little higher than that.

After Python has finished executing the complete `if` statement along with the associated `elif` and `else` clauses, it moves on to the next statement in the block containing the `if` statement. In this case, it is the main block (where execution of the program starts), and the next statement is the `print('Done')` statement. After this, Python sees the end of the program and simply finishes up.

The following code is another example of a conditional statement with more than two branches:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

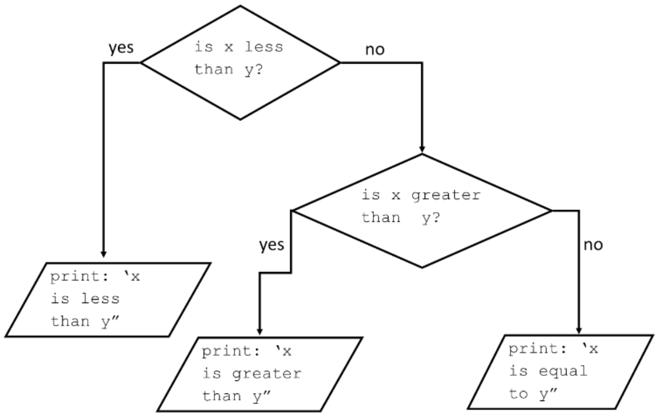


Figure 40: Two or More Branches Conditional Flowchart

`elif` is an abbreviation of "else if". Again, exactly one branch will run. There is no limit on the number of `elif` statements. See the corresponding flowchart in figure 40. If there is an `else` clause, it has to be at the end, but there doesn't have to be one. See the following example:

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch runs and the statement ends. Even if more than one condition is true, only the first `true` branch runs. It is good practice to include the `else` clause as the final branch in the **`if-elif-else`** `statement.`

Practice – Write a program to accept a test score as input and print out the corresponding letter grade. First examine the flowchart (figure 41) of the problem we are solving. Then, using the Python editor, enter the following code (save as `testGrade.py`):

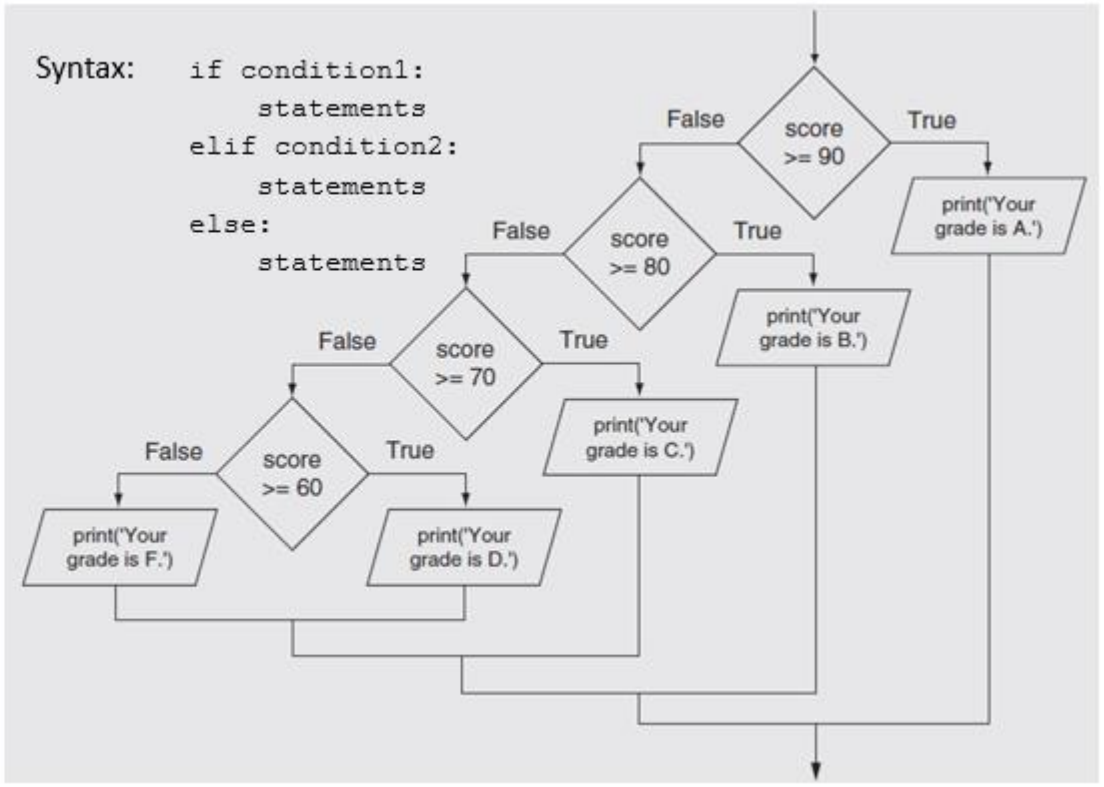
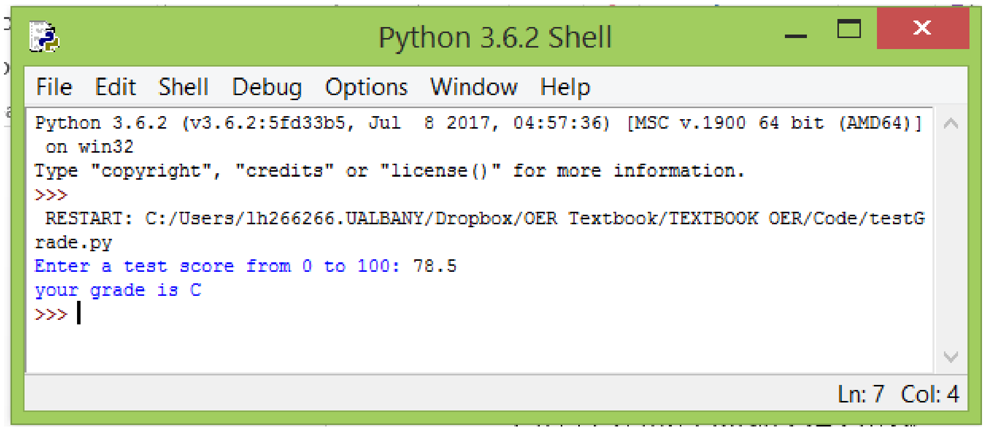

Syntax:
if condition1:
statements
elif condition2:
statements
else:
statements
score >= 90
False
True
print('Your grade is A.')
score >= 80
print('Your grade is B.')
score >= 70
print('Your grade is C.')
score >= 60
print('Your grade is D.')
print('Your grade is F.')

Figure 41 : `if-elif-else` Conditional Flowchart

```
#accept a test score as input and print out the corresponding
letter grade
testScore = float(input("Enter a test score from 0 to 100: "))
# conditional statement to determine and print the test letter
grade
if testScore >= 90:
    print("your grade is A")
elif testScore >= 80:
    print("your grade is B")
elif testScore >= 70:
    print("your grade is C")
elif testScore >= 60:
    print("your grade is D")
else:
    print("your grade is F")
```

Output:



```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:/Users/lh266266.UALBANY/Dropbox/OER Textbook/TEXTBOOK OER/Code/testG
rade.py
Enter a test score from 0 to 100: 78.5
your grade is C
>>>
```

Let us examine how this program works.

Python Statement	Explanation
`testScore = float(input("Enter a test score from 0 to 100: "))`	Accept a numeric test grade from the user as a string and convert the number to a floating point values. Save this value in a variable named "testScore".

`if testScore >= 90:`
`    print("your grade is A")`
	The first branch of this chained conditional is executed. If this condition (the test score is greater than or equal to 90) is `true` then we print the letter grade "A" using the `print` function and end the program. If the condition is `false`, the program continues to the next branch (the first `elif` condition).

`elif testScore >= 80:`
`    print("your grade is B")`
	The second branch of this chained conditional is executed. If this condition (the test score is greater than or equal to 80) is `true` then we print the letter grade "B" using the `print` function and end the program. If the condition is `false`, the program continues to the next branch.
	Note that this statement will not be executed if the test score is above a 90 .This was determined in the first branch (the previous `if` condition). Therefore we know that the score MUST be less than 90.

`elif testScore >= 70:`
`    print("your grade is C")`
	The third branch of this chained conditional is executed. If this condition (the test score is greater than or equal to 70) is `true` then we print the letter grade "C" and end the program. If the condition is `false`, the program continues to the next branch.
	Note that this statement will not be executed if the test score is above a 80 . Therefore we know that the score MUST be less than 80.

`elif testScore >= 60:`
`    print("your grade is D")`
	The fourth branch of this chained conditional is executed. If this condition (the test score is greater than

	or equal to 60) is `true` then we print the letter grade "D" and end the program. If the condition is `false`, the program continues to the next branch. Note that this statement will not be executed if the test score is above a 70 . Therefore we know that the score MUST be less than 70.

`else:`
`    print("your grade is F")`
	The "`else`" is the final branch of the chained conditional statement. In order to execute this branch ALL previous conditions must have been evaluated to "`false`". Note that this statement will not be executed if the test score is above a 60.

One conditional can also be nested within another called **nested conditionals**. Earlier in this section we saw the following example using a chained conditional (**`if-elif-else`**).

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

We could have just as easily written this code using nested conditionals like this:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. It is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

The `print` statement runs only if we make it past both conditionals, so we can get the same effect with the `and` operator:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

For this kind of condition, Python provides even a more concise option:

```
if 0 < x < 10:
    print('x is a positive single-digit number.')
```

Exception Handling

In our programming experience so far we have encountered several kinds of programming run-time exceptions, such as division by zero, attempting to read a variable that has not been defined, and attempting to convert a non-number to an integer. We have seen these and other run-time exceptions immediately terminate a running program.

Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is raised at the point where the error is detected; it may be handled by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred. The Python interpreter raises an exception when it detects a run-time error (such as division by zero).

Python provides a standard mechanism called exception handling that allows programmers to deal with these kinds of run-time exceptions and many more. Rather than always terminating the program's execution (this is called a program "crash"), an executing program can detect the problem when it arises and possibly execute code to correct the issue it in some way.

Exceptions occur when *exceptional situations* occur in your program. Similarly, what if your program had some invalid statements? This is handled by Python which raises its hands and tells you there is an error.

Errors

Consider the simple `print` function call. What if we misspelt `print` as `Print`? Note the capitalization. In this case, Python raises a syntax error. The following output appears in the Python shell. Observe that a `NameError` is raised and also the location of the error detected is printed. This is what an error handler for this error does.

```
>>> Print("hello there!")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    Print("hello there!")
NameError: name 'Print' is not defined
>>> print("hello there!")
hello there!
>>>
```

Exceptions

Let us try to read input from the user. In the Python shell we enter the first line below and hit the Enter key. When the computer prompts for input, instead press [ctrl-c] (this example is with Windows) and see what happens.

```
>>>
>>> text = input("Enter something: ")
Enter something:
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    text = input("Enter something: ")
KeyboardInterrupt
>>>
```

Python raises an error called `KeyboardInterrupt` which basically means it was interrupted when it expected to get user input.

Let us look at another example. Here we will read input from the user and then attempt to convert the string input into an integer.

```
>>> num = input("Enter a number between 1 and 10: ")
Enter a number between 1 and 10: no
>>> int(num)
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    int(num)
ValueError: invalid literal for int() with base 10: 'no'
>>>
```

Python raises an error called `ValueError` which is letting us know that the string `'no'` cannot be converted to an integer value. Programmers can avoid this scenario by checking that a string's value can be converted to a number value.

Another common exception occurs when we try to evaluate a comparison which does not make sense. Consider the following Python conditional statements executed in the Shell window:

```
>>> 12 > 5
True
>>> 12 == 5
False
>>> 12 < 'a'
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    12 < 'a'
TypeError: '<' not supported between instances of 'int' and 'str'
>>>
```

Python raises an error called `TypeError` which is letting us know that the condition `12 < 'a'` cannot be performed.

Handling Exceptions

Essentially, exceptions are events that modify program's flow, either intentionally or due to errors. They are special events that can occur due to an error, e.g. trying to divide a number by zero. Exceptions, by definition, don't occur very often; hence, they are the "exception to the rule".

Exceptions are everywhere in Python. Virtually every module in the standard Python library uses them, and Python itself will raise them in a lot of different circumstances. One use of exceptions is to catch an error and allow the program to continue working instead of crashing ungracefully.

This is the most common way to use exceptions. When programming with the Python command line interpreter (in the Shell window), you don't need to worry about catching exceptions. Your program is usually short enough to not be hurt too much if an exception occurs. Plus, having the exception occur at the command line is a quick and easy way to tell if your code logic has a problem. However, if the same error occurred in your saved .py file program, it will crash (fail) and stop working.

Exceptions can be thought of as a special form of the `if-else` statements. You can realistically do the same thing with `if` blocks as you can with exceptions. However, as already mentioned, exceptions aren't processed until they occur; `if` blocks are processed all the time. Proper use of exceptions can help the performance of your program. The more infrequent the error might occur, the better off you are to use exceptions; using `if` blocks requires Python to always test extra conditions before continuing. Exceptions also make code management easier: if your programming logic is mixed in with error-handling `if` statements, it can be difficult to read, modify, and debug your program.

We can handle exceptions using the `try-except-else` statement. Briefly, `try` creates a block that attempts to perform an action. If that action fails, the `except` block catches any exception that is raised and notifies the user that an error occurred, then stops executing the program. If the code within the `try` block does not produce an exception, the program's execution continues with code in the `else` block.

Practice - Here is a simple program that uses exception processing. It simply produces the quotient of 2 numbers. Try entering, saving and running the following program (`exceptionHandling.py`).

```
# exception handling
first_number = input ( "Enter the first number:  ") #gets input from
keyboard
sec_number = input ( "Enter the second number:  " )
try :
    num1 = float( first_number ) #try turning keyboard input into floats
    num2 = float( sec_number )
    result = num1/num2           #try dividing num1 by num2
except:
    print("Error: enter numbers only; num2 cannot equal zero")
else :
    print(str(num1) + "/" + str(num2) + "=" + str(result))
```

The output from executing this program, three separate times, is shown below.

```
RESTART: C:/Users/lh266266.UALBANY/Dropbox/OER Textbook/TEXTBOOK
OER/Code/exceptionHandling.py
Enter the first number:  33
Enter the second number:  22
33.0/22.0=1.5
>>>
 RESTART: C:/Users/lh266266.UALBANY/Dropbox/OER Textbook/TEXTBOOK
OER/Code/exceptionHandling.py
Enter the first number:  33
Enter the second number:  twenty
Error: enter numbers only; num2 cannot equal zero
>>>
 RESTART: C:/Users/lh266266.UALBANY/Dropbox/OER Textbook/TEXTBOOK
OER/Code/exceptionHandling.py
Enter the first number:  33
Enter the second number:  0
Error: enter numbers only; num2 cannot equal zero
>>>
```

In this example we simply print out a message to the user in the `except` block and then gracefully terminate our program. We use the `else` statement at the end to perform the rest of the program logic if all goes well. As stated before, the whole try block could also have been written as `if/else` statements but that would have required Python to process each statement to see if they matched. By using exceptions, the "default" case is assumed to be true until an exception actually occurs. This speeds up processing.

It's better to include error-checking, such as exceptions, in your code as you program rather than as an afterthought. A special "category" of programming involves writing test cases (we wrote test cases for our algorithms in Unit 1!) to ensure that most possible errors are accounted for in the code, especially as the code changes or new versions are created. Recall the test cases we were introduced to when writing our algorithms in Unit 1. By planning ahead and putting exception handling into your program at the outset, you ensure that problems are caught before they can cause problems.

Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is raised at the point where the error is detected; it may be handled by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred. The Python interpreter raises an exception when it detects a run-time error (such as division by zero).

Practice with Handling Exceptions in our Programs

Let us review some examples of handling exceptions in some of our programs we have written.

Practice #1 - Calculating the distance between two points:

```
# calculates the distance between two points

import math

print("This program calculates the distance between two points.")
print() #print a blank line

x1 = float(input("Enter the x for the first point: "))
y1 = float(input("Enter the y for the first point: "))
print()
x2 = float(input("Enter the x for the second point: "))
y2 = float(input("Enter the y for the second point: "))

distance = math.sqrt((x2-x1)**2 + (y2-y1)**2)

print()
print("The distance between the points is", distance)
```

The output when executing this program with non-numeric input is seen below:

```
>>>
 RESTART:
C:/Users/lh266266.UALBANY/distanceBetweenPointsExceptionHandling.py
This program calculates the distance between two points.
Enter the x for the first point: 17
Enter the y for the first point: 23
Enter the x for the second point: 88
Enter the y for the second point: 9p
Traceback (most recent call last):
  File "C:\Users\lh266266.UALBANY\distanceBetweenPoints.py", line 12,
in <module>
    y2 = float(input("Enter the y for the second point: "))
ValueError: could not convert string to float: '9p'
>>>
```

Rewrite this program to include exception handling to catch the `ValueError` when we try and convert our user input into floating point numbers.

```
# Calculates the distance between two points

import math

print("This program calculates the distance between two points.")
print()
try:
    x1 = float(input("Enter the x for the first point: "))
    y1 = float(input("Enter the y for the first point: "))
    print()
    x2 = float(input("Enter the x for the second point: "))
    y2 = float(input("Enter the y for the second point: "))
except:
    print("Error: enter numeric values only")
else:
    distance = math.sqrt((x2-x1)**2 + (y2-y1)**2)
    print()
    print("The distance between the points is", distance)
```

The output when executing the revised program with non-numeric input is far more user friendly!

```
>>>
 RESTART:
C:/Users/lh266266.UALBANY/distanceBetweenPointsExceptionHandling.py
This program calculates the distance between two points.
Enter the x for the first point: 17
Enter the y for the first point: 23
Enter the x for the second point: 88
Enter the y for the second point: 9p
Error: enter numeric values only
>>>
```

Practice #2 – Guess the Number:

```
number = 23
guess = int(input('Enter an integer : '))
if guess == number:
    print('Congratulations, you guessed it.')
    print('(but you do not win any prizes!)')
elif guess < number:
    print('No, it is a little higher than that')
else:
    print('No, it is a little lower than that')
print('Done')
```

The output when executing this program with non-numeric input is seen below:

```
>>>
 RESTART: C:/Users/lh266266.UALBANY/ifElse.py
Enter an integer : 8g6
Traceback (most recent call last):
  File "C:\Users\lh266266.UALBANY\Dropbox\OER Textbook\TEXTBOOK
OER\Code\ifElse.py", line 2, in <module>
    guess = int(input('Enter an integer : '))
ValueError: invalid literal for int() with base 10: '8g6'
>>>
```

Rewrite this program to include exception handling to catch the `ValueError` when we try and convert the user input into an integer.

```
# Guess my number

number = 23        #this is 'my number'
try:
    guess = int(input('Enter an integer : '))
except:
    print("ERROR: enter whole numbers only")
else:
    if guess == number:
        print('Congratulations, you guessed it.')
        print('(but you do not win any prizes!)')
    elif guess < number:
        print('No, it is a little higher than that')
    else:
        print('No, it is a little lower than that')

    print('Done')
```

The output when executing the revised program with non-numeric input gracefully ends rather than crashing!

```
>>>
 RESTART: C:/Users/lh266266.UALBANY/ifElseExceptionHandling.py
Enter an integer : 88m
ERROR: enter whole numbers only
>>>
```

Iteration

This section is about iteration, which is the ability to run a block of statements repeatedly. Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. In a computer program, repetition is also called iteration.

The `while` Statement

A `while` statement is an iterative control statement that repeatedly executes a set of statements based on a provided Boolean expression (condition). All iterative control needed in a program can be achieved by use of the `while` statement, though, as we will see later, other iterative control statements can be more useful for solving some computational problems.

Here is a version of a countdown function that uses a `while` statement:

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Blastoff!')
```

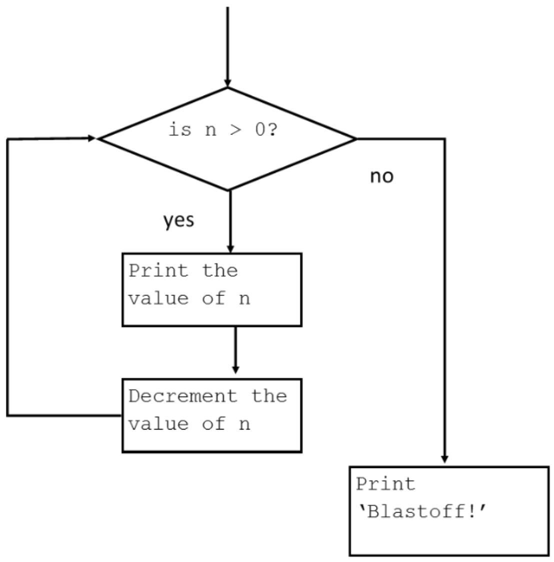


Figure 42: While Loop Flowchart

You can almost read the `while` statement as if it were English. It means, "While n is greater than 0, display the value of n and then decrement n. When you get to 0, display the word Blastoff!"

Here is the flow of execution for a while statement:

1. Determine whether the condition is true or false.
2. If false, exit the while statement and continue execution at the next statement.
3. If the condition is true, run the body and then go back to step 1.

This type of flow is called a loop because the third step loops back around to the top.

The body of the loop should change the value of one or more variables so that the condition becomes false eventually and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite** loop.

In the case of the countdown function, we can prove that the loop terminates: if n is zero or negative, the loop never runs. Otherwise, n gets smaller each time through the loop, so eventually we have to get to 0.

Practice 1: Try executing the following program which uses the countdown function (`countdown.py`).

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Blastoff!')
num = input("enter a positive number to countdown from: ")
try:
    num=int(num)
except:
    print("ERROR: enter a whole number")
else:
    countdown(num)
```

An example of the output after execution:

```
RESTART: C:/Users/lh266266.UALBANY/Code/countdown.py
enter a number to countdown from: 9
9
8
7
6
5
4
3
2
1
Blastoff!
```

The variable **`n`** in the `countdown.py` program is referred to as a *counter*. A *counter* variable is used to count something, usually initialized to zero and then incremented (or decremented) and is commonly used in programs.

The `while` statement allows you to repeatedly execute a block of statements as long as a condition is true. A `while` statement is an example of what is called a *looping* statement. A `while` statement can have an optional else clause. Let us look at the following program (`while.py`):

```
number = 23
running = True

while running:
    guess = int(input('Enter an integer : '))

    if guess == number:
        print('Congratulations, you guessed it.')
        # this causes the while loop to stop
        running = False
    elif guess < number:
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')
else:
    print('The while loop is over.')
    # Do anything else you want to do here

print('Done')
```

In this program, we are still playing the guessing game, but the advantage is that the user is allowed to keep guessing until he guesses correctly - there is no need to repeatedly run the program for each guess, as we have done in the previous section. This aptly demonstrates the use of the `while` statement.Let us examine more closely the code in this program (`while.py`).

Python Statement	Explanation

`number = 23`
`running = True`
	We set the variable `number` to a number we want; here we choose the number '23'.The variable running is referred to as a 'flag' variable. Flag variables go by many different names, but they all represent a variable (usually of `Boolean` type) that can only be one of two states, `True` or `False`.

`while running:`	This where the loop starts. First, we check if the variable `running` is `True` and then proceed to execute the corresponding while-block. After this block is executed, the condition is again checked which in this case is the `running` variable. If it is true, we execute the while-block again, else we continue to execute the optional else-block and then continue to the next statement.

`guess = int(input('Enter an integer : '))`
`if guess == number:`
`    print('Congratulations, you guessed it.')`
`    running = False`
`elif guess < number:`
`    print('No, it is a little higher than that.')`
	This is the 'body' of the entire loop.
	The user is asked to enter an integer number which is converted into an integer and saved in the variable guess.
	Next, we compare the guess of the user with the number we have chosen. If they are equal, we print a success message and set our flag variable,

`else:`
`    print('No, it is a little lower than that.')`
	`running`, to `False`. This will soon cause the while loop to stop.
	Then, we check if the guess is less than the number, and if so, we inform the user that they must guess a little higher than that.
	Finally we check if the guess is greater than the number, and if so, we inform the user that they must guess a little lower than that.

`print('Done')`	Once we have exited the `while` loop we print this single statement to indicate to the user that the guessing game is over.

The following is a sample of output after running this program:

```
Enter an integer : 5
No, it is a little higher than that.
Enter an integer : 19
No, it is a little higher than that.
Enter an integer: 25
No, it is a little lower than that.
Enter an integer: 24
No, it is a little lower than that.
Enter an integer: 23
Congratulations, you guessed it.
The while loop is over.
Done
>>>
```

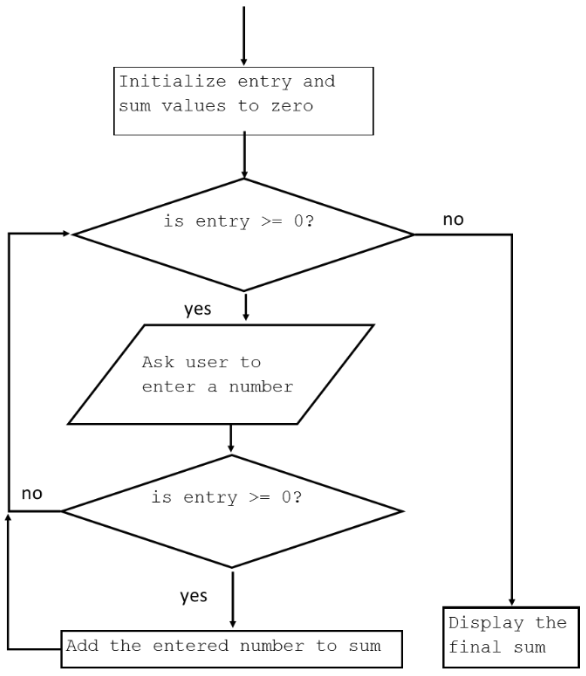
This is an example of an *interactive loop* because it allows the user to repeat something (in this example the user is guessing a number) on demand.

Practice 2: The following program allows a user to enter any number of nonnegative integers. When the user enters a negative value, the program no longer accepts input, and it displays the sum of all the nonnegative values. If a negative number is the first entry, the sum is zero. This is also an example of *interactive loop*. [`addNonNegatives.py`]

```
# Allow the user to enter a sequence of nonnegative integers to sum.
entry = 0                             # Ensure the loop is entered
sum = 0                               # Initialize sum
print("Enter positive numbers to sum (entering a negative number ends
the program).")
while entry >= 0:                     # A negative number exits the loop
    entry = int(input("number: "))    # Get the value
    if entry >= 0:                    # Is number nonnegative?
        sum = sum + entry
print("Sum =", sum)                   # Display the sum
```

```
RESTART:C:/Users/lh266266.UALBANY/Code/addNonNegatives.py
Enter positive numbers to sum (entering a negative number ends the program).
number: 8
number: 9
number: 1
number: 2
number: -9
Sum = 20
```

Figure 43: Sample Output from addNonNegatives.py



Initialize entry and sum values to zero
is entry >= 0?
no
yes
Ask user to enter a number
no
is entry >= 0?
yes
Add the entered number to sum
Display the final sum

Figure 44: While Loop Flowchart

Let us examine more closely the code in this program.

Python Statement	Explanation
`entry = 0`	In the beginning we initialize `entry` to zero for the sole reason that we want the `condition entry >= 0` of the while statement to be `true` initially. If we fail to initialize `entry`, the program will produce a run-time error when it attempts to compare `entry` to zero in the while condition. The `entry` variable holds the number entered by the user. Its value can change each time through the loop.
`sum = 0`	The variable `sum` is known as an *accumulator* because it accumulates each value the user enters. We initialize `sum` to zero in the beginning because a value of zero indicates that it has not accumulated anything. If we fail to initialize `sum`, the program will generate a run-time error when it attempts to use the + operator to modify the (non-existent) variable.

`print("Enter positive numbers to sum (entering a negative number ends the program).")`	This statement provides instructions for this app.

`while entry >= 0:`
`    entry = int(input("number: "))`
`    if entry >= 0:`
`        sum = sum + entry`
	Within the loop we repeatedly add the user's input values to `sum`. When the loop finishes (because the user entered a negative number), `sum` holds the sum of all the nonnegative values entered by the user.

`print("Sum =", sum)`	Once we have exited the `while` loop we print this single statement to indicate to the user what the sum of all the numbers entered is.

Practice 3: Now let us rewrite this program to add additional functionality. In addition to finding the sum of nonnegative values we will calculate the average. [`avgNonNegatives.py`]

```
# Allow the user to enter a sequence of nonnegative integers to sum and find
the average.
entry = 0                                  # Ensure the loop is entered
sum = 0                                    # Initialize sum
count = 0                                  # Initialize count
print("Enter positive numbers to sum (entering a negative number ends the
program).")
while entry >= 0:                          # A negative number exits the loop
    entry = int(input("number: "))         # Get the value
    if entry >= 0:                         # Is number nonnegative?
        sum = sum + entry             # Only add it if it is nonnegative
        count = count + 1             #increment the count of numbers entered
print("Sum =", sum)                   # Display the sum
average = sum / count
print("Average =", average)
```

The program needs not only an accumulator variable (i.e. `sum`) but a counter variable also (named `count`) to keep track of how many numbers have been entered. We can then calculate the average by dividing the sum by the count.

```
RESTART: C:/Users/lh266266.UALBANY/Code/avgNonNegatives.py
Enter positive numbers to sum (entering a negative number ends the program).
number: 12
number: 55
number: 31
number: -11
Sum = 98
Average = 32.666666666666664
```

Figure 45: Sample Output from avgNonNegatives.py

The `break` Statement

Sometimes you don't know it's time to end a loop until you get half way through the body. In that case you can use the `break` statement to jump out of the loop.

For example, suppose you want to take input from the user until they type `done`. You could write:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)

print('Done!')
```

The loop condition is `True`, which is always true, so the loop runs until it hits the `break` statement.

Each time through, it prompts the user with an angle bracket. If the user types done, the `break` statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

```
> hello
hello
> done
Done!
```

This way of writing `while` loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens").

Note that the `break` statement works with the `for` loop as well.

The `continue` Statement

The `continue` statement is used to tell Python to skip the rest of the statements in the current loop block and to continue to the next iteration of the loop.

Let us examine the following example (save as `continue.py`):

```
# user is to enter a string of at minimum 3 words
# to end the user is to enter the word 'quit'
while True:
    data = input('Enter at least 3 words: ')
    if data == 'quit':
        break
    if len(data) < 3:
        print('Too small')
        continue
    print('Input (data) is of sufficient length')
    # Do other kinds of processing here...
```

```
>>>
 RESTART:C:/Users/lh266266.UALBANY/Code/continue.py
Enter at least 3 words: hi
Too small
Enter at least 3 words: how are you
Input (data) is of sufficient length
Enter at least 3 words: quit
>>>
```

In this program, we accept input from the user, but we process the input string only if it is at least 3 words long or the user enters the word ‘quit’. So, we use the built-in `len` function to get the length and if the length is less than 3, we skip the rest of the statements in the block by using the `continue` statement. Otherwise, the rest of the statements in the loop are executed, doing any kind of processing we want to do here. When the user enters the word ‘`quit`’ we exit the loop using the `break` statement, then end the program. This is an example of an *input validation loop*. In an *input validation loop* (commonly designed using the `while` loop) the program inspects user input before it is processed by the program and if input is invalid it prompts user to enter correct data.

Note that the `continue` statement works with the `for` loop as well.

The `for...in` Statement

The `for..in` statement is another looping statement which iterates over a sequence of objects i.e. go through each item in a sequence. Recall that a sequence is just an ordered collection of items. Let us look at the following program, `for.py`.

```
for i in range(1, 5):
    print(i)
else:
    print('The for loop is over')
```

This is what the output of the program looks like in the Python shell after it runs:

```
1
2
3
4
The for loop is over
>>>
```

In this program, we are printing a sequence of numbers. We generate this sequence of numbers using the built-in `range` function.

What we do here is supply it two numbers and `range` returns a sequence of numbers starting from the first number and up to the second number. For example, `range(1,5)` gives the sequence `[1, 2, 3, 4]`. By default, `range` takes a step count of 1. If we supply a third number to `range`, then that becomes the step count. For example, `range(1,5,2)` gives `[1,3]`. Remember that the `range` extends up to the second number i.e. it does not include the second number.

The `for` loop then iterates over this range - `for i in range(1,5)` is equivalent to `for i in [1, 2, 3, 4]` which is like assigning each number (or object) in the sequence to `i`, one at a time, and then executing the block of statements for each value of `i`. In this case, we just print the value in the block of statements.

The general form of the range expression is `range( begin,end,step )` where

- `begin` is the first value in the range; if omitted, the default value is 0
- `end` is one past the last value in the range; the end value is always required and may not be omitted
- `step` is the amount to increment or decrement; if the step parameter is omitted, it defaults to 1 (counts up by ones)

`begin`, `end`, and `step` must all be integer expressions; floating-point expressions and other types are not allowed. The arguments in the range expression may be literal numbers (like 10), variables (like `x`, if `x` is equal to an integer), and arbitrarily complex integer expressions.

The `range` expression is very flexible. Consider the following loop that counts down from 21 to 3 by threes:

```
for n in range(21, 0, -3):
    print(n, end=' ')
```

It prints:

```
21 18 15 12 9 6 3
```

Thus `range(21, 0, -3)` represents the sequence `21;18;15;12;9; 3`.

The expression `range(1000)` produces the sequence `0;1;2; : : : ; 999`.

The following code computes and prints the sum of all the positive integers less than 100:

```
sum = 0 # Initialize sum
for i in range(1, 100):
    sum += i
    print(sum)
```

Remember that the `else` block is optional. When included, it is always executed once after the `for` loop is over unless a `break` statement is encountered.

So far we have used the `for` loop to iterate over integer sequences because this is a useful and common task in developing apps. The `for` loop, however, can iterate over any iterable object, such as a string.

We can use a `for` loop to iterate over the characters that comprise a string. The following program uses a `for` loop to print the individual characters of a string (`printLetters.py`).

```
word = input('Enter a word: ')
for letter in word:
    print(letter)
```

Sample output:

```
RESTART: C:/Users/lh266266.UALBANY/Code/printLetters.py
Enter a word: hello
h
e
l
l
o
>>>
```

Practice 1: Write a program to solve the following problem. [`printWords.py`]

Problem: Ask the user how many words they want to enter then print the words entered by the user on one line separated by spaces.

Input: Number of words to be entered; this value must be a positive integer greater than zero.

```
# Ask the user how many words they want to enter
# then print the words together on one line.
sentence = ""
num = input("How  many words do you want to enter? ")
try:
    num=int(num)
except:
    print ("ERROR: enter a number")
else:
    if num > 0:  #check for a positive number
        for i in range(num):
            word = input("enter a word: ")
            sentence = sentence + " " + word
        print(sentence)
    else:
        print("Will only accept a positive integer number. Ending program")
```

Note the error checking performed in the program:

- Ensures the user only enters integer numbers as input
- Ensures the user enters a positive integer

Practice 2: Write a program to solve the following problem. [`numWords.py`]

Problem: Ask the user to enter in a complete sentence and display the number of words in the sentence.

Unlike the example where we used a `for` loop to iterate over the characters of a word, we need to use a string object's method to break the string into words. When you need to break a large string down into smaller chunks, or strings, you use the `split()` string method. By default, `split()` takes whitespace as the delimiter. Try executing this program (`numWords.py`).

```
# Ask the user to enter in a complete sentence and display the number
of words in the sentence.
sentence = input("please enter an awe inspiring sentence or two: ")
words = sentence.split()
print("you entered", len(words), "words")
print("here are the individual words you entered:", words)
```

Example output:

```
please enter an awe inspiring sentence or two: When I need to build a web app,
I reach for the Python programming language. When I need to automate some small
task on my system, I reach for Python. Python rocks!
you entered 32 words
here are the individual words you entered: ['When', 'I', 'need', 'to', 'build',
'a', 'web', 'app,', 'I', 'reach', 'for', 'the', 'Python', 'programming',
'language.', 'When', 'I', 'need', 'to', 'automate', 'some', 'small', 'task',
'on', 'my', 'system,', 'I', 'reach', 'for', 'Python.', 'Python', 'rocks!']
>>>
```

Once you have used `split` to break the string into a list of words, you can use the index operator (square bracket) to look at a particular word in the list.

```
>>> sentence = "Python is a general purpose programming language named
after Monty Python."
>>> words = sentence.split()
>>> print(words)
['Python', 'is', 'a', 'general', 'purpose', 'programming', 'language',
'named', 'after', 'Monty', 'Python.']
>>> print(words[9], words[3], words[6])
Monty general language
>>>
```

Enhance this program to loop through each word in the list named `words` and print each word in uppercase on its own line (`uppercaseWords.py`).

```
# Ask the user to enter in a complete sentence and display
each work in uppercase.
sentence = input("please enter an awe inspiring sentence or
two: ")
words = sentence.split()
print("you entered", len(words), "words")
print("here are the individual words you entered:")
for word in words:
    print(word.upper())
```

Example output:

```
please enter an awe inspiring sentence or two: Python is a general purpose
programming language named after Monty Python.
you entered 11 words
here are the individual words you entered:
PYTHON
IS
A
GENERAL
PURPOSE
PROGRAMMING
LANGUAGE
NAMED
AFTER
MONTY
PYTHON.
>>>
```

How to choose between the `for` and `while` loop?

Use a `for` loop if you know, before you start looping, the maximum number of times that you'll need to execute the body. For example, if you're traversing a list of elements, you know that the maximum number of loop iterations you can possibly need is "all the elements in the list". Or if you need to print the 12 times table, we know right away how many times the loop will need to run.

So any problem like "iterate this weather model for 1000 cycles", or "search this list of words", "find all prime numbers up to 10000" suggest that a `for` loop is best.

By contrast, if you are required to repeat some computation until some condition is met, and you cannot calculate in advance when (of if) this will happen, as we did in this 3n + 1 problem, you'll need a `while` loop.

We call the first case **definite iteration** — we know ahead of time some definite bounds for what is needed. The latter case is called **indefinite iteration** — we're not sure how many iterations we'll need — we cannot even establish an upper bound!

Nested Loops

Just like with `if` statements, `while` and `for` blocks can contain arbitrary Python statements, including other loops. A loop can therefore be nested within another loop. To see how nested loops work, consider a program that prints out a multiplication times tables (`multiplication.py`).

```
#multiplcation tables
try:
    table = int(input("which multiplication times table do you want to
print? (choose from 1 to 12) "))
    x = table
except:
    print("ERROR: enter a whole number")
else:
    if table > 0 and table < 13:
        for y in range(1, 13):
            print (x,'*', y,'=', x*y)
    else:
          print ("ERROR: multiplication tables can be generated from 1 to 12
only")
```

Let us examine more closely the code in this program.

Python Statement	Explanation

`try:`
`    table = int(input("which multiplication times table do you want to print? (choose from 1 to 12) "))`
`    x = table`
`except:`
`    print("ERROR: enter a whole number")`
`else:`
`     if table > 0 and table < 13:`
	We use the `try-except-else` block to catch the error of non-numeric input from the user.
	The `if` block following the `else` block assures that the user is entering a number in the correct range of 1 to 12.

`for y in range(1, 13):`
`     print (x,'*', y,'=', x*y)`
	The `for` loop is generating the multiplication times table requested from the user.The variable `x` represents the times-table the user requested and the variable `y` provides each entry in the times table. We use a loop to print the contents of each row. The outer loop controls how many total rows the program prints, and the inner loop, executed in its entirety each time the program prints a row, prints the individual elements that make up a row.

`print("Enter positive numbers to sum (entering a negative number ends the program).")`	This statement provides instructions for this app.

`else:`
`      print ("ERROR: multiplication tables can be generated from 1 to 12 only")`
	This is the end of the program. This block, which prints and error message, only executes if the user did not enter a number in the correct range of 1 to 12.

Example output:

```
RESTART: C:/Users/lh266266.UALBANY/Code/multiplication.py
which multiplication times table do you want to print? (choose from 1
to 12) 5
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
5 * 11 = 55
5 * 12 = 60
>>>
```

Now let us consider adding to this program to generate all the multiplication times tables from 1 to 12. This is easily done with nested loops.

```
#multiplcation times tables tables
for x in range(1, 13):
    for y in range(1, 13):
        print (x,'*', y,'=', x*y)
```

The output after execution of this 3-lines-of-code program generates 144 lines of output!

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
1 * 10 = 10
1 * 11 = 11
1 * 12 = 12
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20
2 * 11 = 22
2 * 12 = 24
:
:
12 * 1 = 12
12 * 2 = 24
12 * 3 = 36
12 * 4 = 48
12 * 5 = 60
12 * 6 = 72
12 * 7 = 84
12 * 8 = 96
12 * 9 = 108
12 * 10 = 120
12 * 11 = 132
12 * 12 = 144
```

Basic File Processing

While a program is running, its data is stored in random access memory (RAM) on the computer it is execution on. RAM is fast and inexpensive, but it is also volatile, which means that when the program ends, or the computer shuts down, the data in RAM disappears. To make data available the next time the computer is turned on and the program is started, it has to be written to a non-volatile storage medium, as a file. By reading and writing files, programs can save information between program runs.

So far, the data we have used in this book have all been either coded right into the program, or have been entered by the user. In real life data reside in files.

For our purposes, we will assume that our data files are text files–that is, files filled with characters. The Python programs that you write are stored as text files as are any HTML webpage files. We can create these files in any of a number of ways. For example, we could use a text editor to type in and save the data. We could also download the data from a website and then save it in a file. Regardless of how the file is created, Python will allow us to manipulate the contents. Note that text files have an End-Of-Line (EOL) character to indicate each line's termination.

Reading and Writing Text Files

In Python, we must open files before we can use them and close them when we are done with them. As you might expect, once a file is opened it becomes a Python object just like all other data. The following table shows the functions and methods that can be with files.

Function/Method Name	**Use**	**Explanation**
open	`f1 = open('workfile.txt', 'r')`	Open a file called *workfile.txt* and use it for reading. This will return a reference to a file object assigned to the variable `f1`.
open	`f1 = open('workfile.txt', 'w')`	Open a file called *workfile.txt* and use it for writing. This will also return a reference to a file object assigned to the variable `f1`.
close	`f1.close()`	When a file use is complete, the file must be closed. This will free up any system resources taken up by the open file.
read	`data = f1.read()`	Read the entire contents of the file `f1` and assign it to the variable `data`.
write	`f1.write(string)`	Write the contents of *string* to the file `f1`, returning the number of characters written.
readline	`data = f1.readline()`	Read a single line from the file `f1`; a newline character (`\n`) is left at the end of the string.

Table 7: File Functions and Methods

Python file object attributes provide information about the file and file state. The following table shows how to identify some of the object's attributes.

Attribute	**Use**	**Explanation**
file.closed	`print("Closed or not : ", file1.closed)`	Returns true if `file1` is closed, false otherwise.
file.mode	`print("Opening mode : ", file1.mode)`	Returns access mode with which `file1` was opened.
file.name	`print("Name of the file: ", file1.name)`	Returns name of `file1`.

Table 8: File Object Attributes

The statement

```
file1 = open('myfile.txt', 'r')
```

creates and returns a file object in the variable named `file1`. The first argument to open is the name of the file, and the second argument is a mode (in this example the mode is to read the file). This statement will attempt to read the content of the text file named `myfile.txt`. If the file does not exist or the user of the program does not have adequate permissions to open the file, the `open` function will raise an exception.

Similar to handling an error of invalid user input we need to use the try-except-else block to handle trying to read from a file which does not exist The following is an example:

```
try:
    file1 = open('myfile.txt', 'r')
except:
    print("ERROR: unable to open or locate the file")
# Do other kinds of processing here...
```

The statement

```
file2 = open('myfile.txt', 'w')
```

creates and returns a file object in the variable named `file2` which will write data to the text file named `myfile.txt`. If the file does not exist, the function creates the file on disk; no exceptions are raised when writing to a file. If a file by that name currently exists, new data will replace the current data stored in the file. This means any pre-existing data in the file will be lost.

Once you have a file object capable of writing (opened with 'w') you can save data to the file associated with that file object using the `write` method. For a file object named `file2`, the statement

```
file2.write('data')
```

stores the string 'data' to the file. The three statements

```
file2.write('data')
file2.write('compute')
file2.write('process')
```

writes the text `'datacomputeprocess'` to the file. If our intention is to retrieve the three separate original strings, we must add delimiters to separate the pieces of data. Newline characters serve as good delimiters:

```
file2.write('data\n')
file2.write('compute\n')
file2.write('process\n')
```

This places each word on its own line in the text file. The advantage of storing each piece of data on its own line of text is that it makes it easier to read the data from the file with a `for` statement. If `file2` is a file object created for reading, the following code:

```
for line in file2:
    print(line.strip())
```

reads in each line of text from the file and prints it out. The variable `line` is a string, and we use the `strip` method to remove the trailing newline ('`\n`') character.

We also can read the contents of the entire file into a single string using the file object's `read` method:

```
contents = file2.read()
```

Given the text file from above, the code

```
in = open('compterms.txt', 'r')
str = in.read()
```

assigns to `str` the string `'data\ncompute\nprocess\n'`.

The `open` method opens a file for reading or writing, and the `read`, `write`, and other such methods enable the program to interact with the file. When the executing program is finished with its file processing it must call the `close` method to close the file properly. Failure to close a file can have serious consequences when writing to a file, as data meant to be saved could be lost. Every call to the open function should have a corresponding call to the file object's `close` method.

Practice - Following are two simple programs designed to write data to a file and then read and print each line of a file. Try entering, saving and running the following programs (`writeFile.py` and `readFile.py`).

```
file1 = open("compterms.txt", 'w')  #Create a file to write to
# write data to the opened file
file1.write('data\n')
file1.write('compute\n')
file1.write('process\n')
file1.write('file format\n')
file1.write('gigabyte\n')
file1.close()   # Close the file after processing data
```

Figure 46: writeFile.py

```
try:
    file1 = open('compterms.txt','r') # try to find & open the file
except:
    print('ERROR: unable to open or locate the file')
else:
    for line in file1:       # Read each line as text
        print(line.strip()) # Remove trailing newline character and
print the line
    file1.close()            # Close the file after processing data
```

Figure 47: readFile.py

This is the output after running this program:

```
RESTART: C:/Users/lh266266.UALBANY/Code/readFile.py
data
compute
process
file format
gigabyte
>>>
```

We can also read and process each line of data in a file by using the readline() method. Try entering, saving and running the following programs (`readFile2.py`). The output is the same.

```
try:
    file1 = open('compterms.txt','r') # try to find & open the file
except:
    print('ERROR: unable to open or locate the file')
else:
    line = file1.readline()     # Read the first line
    while line:                 # Enter the loop as long as there is a line of
data to read
        print(line.strip())     # Remove trailing newline character and print
the line
        line = file1.readline() # Read the next line
    file1.close()               # Close the file after processing data
```

Processing "Big Data"

Big data is large, and often times complex, data sets that need to be processed into useful information. Python has emerged over the past few years as a leader in data science programming including the analysis and visualization of large data sets.

Data sets grow rapidly - in part because they are increasingly gathered by cheap and numerous information-sensing internet of things devices such as mobile devices, aerial (remote sensing), software logs, cameras, microphones, radio-frequency identification (RFID) readers and wireless sensor networks. The world's technological per-capita capacity to store information has roughly doubled every 40 months since the 1980s; as of 2012, every day 2.5 exabytes (2.5×1018) of data are generated. Based on an IDC (International Data Corporation) report prediction, the global data volume will grow exponentially from 4.4 zettabytes to 44 zettabytes between 2013 and 2020. By 2025, IDC predicts there will be 163 zettabytes of data.[21]

An example of a successful business that uses python for collecting and analyzing data is ForecastWatch. ForecastWatch.com is in the business of rating the accuracy of weather reports from companies such as Accuweather, MyForecast.com, and The Weather Channel. Over 36,000 weather forecasts are collected every day for over 800 U.S. cities, and later compared with actual climatological data. These comparisons are used by meteorologists to improve their weather forecasts, and to compare their forecasts with others. They are also used by consumers to better understand the probable accuracy of a forecast.[22]

Figure 48: Data-Information (www.maxpixel.net/Circle-Know-Arrangement-Data-Information-Learn-229113)

We will use real data from open source data sets (that is saved as text files) for programming practice.

Practice

Problem Statement: Using the text file, `NYScoastalFlooding.txt`, identify each of the New York State counties (zones) effected by coastal flooding during the month of August within the dataset provided.

The first step in finding a solution for this problem is to review and understand the format of the data set provided. Open the text in a text editor and note the field headings in the first line of the file. The following is a small sample from the `NYScoastalFlooding.txt` file.

[21] Wikipedia contributors. "Big data." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 4 Jul. 2018. Web. 12 Jul. 2018.

[22] *Python Software Foundation.* Python Success Stories, https://www.python.org/about/success/forecastwatch/. Accessed July 12, 2018

```
YEAR, MONTH_NAME, EVENT_TYPE, CZ_TYPE,CZ_FIPS, CZ_NAME
2017, March, Coastal Flood, Z, 79, NORTHEAST SUFFOLK
2017, March, Coastal Flood, Z, 179, SOUTHERN NASSAU
2017, March, Coastal Flood, Z, 71, SOUTHERN WESTCHESTER
2017, March, Coastal Flood, Z, 81, SOUTHEAST SUFFOLK
2017, October, Coastal Flood, Z, 3, MONROE
2017, October, Coastal Flood, Z, 5, NORTHERN CAYUGA
2017, October, Coastal Flood, Z, 4, WAYNE
2017, October, Coastal Flood, Z, 6, OSWEGO
2017, January, Coastal Flood, Z, 79, NORTHEAST SUFFOLK
2017, October, Coastal Flood, Z, 1, NIAGARA
2017, January, Coastal Flood, Z, 179, SOUTHERN NASSAU
```

We see that the data set has six fields:

1. year: Four digit year for the event in this record
2. month_name: Name of the month for the event in this record (spelled out; not abbreviated)
3. event_type spelled out; not abbreviated
4. cz_type: Indicates whether the event happened in a (C) county/parish, (Z) zone or (M) marine
5. cz_fips: The county FIPS number is a unique number assigned to the county by the National Institute for Standards and Technology (NIST) or NWS Forecast Zone Number
6. cz_name: County/Parish, Zone or Marine Name assigned to the county FIPS number or NWS Forecast Zone

We also notice that all but the first field on each line has a blank space before it (this is important!).

We will break this problem into smaller pieces. Let us start with a program that reads each of the records and keeps a count of the number of records in the file (`dataRecordNumb.py`). This program is a modification of our `readFile.py` program .

```
try:
    file1 = open('NYScoastalFlooding.txt ','r') # try to find & open 
the file
except:
    print('ERROR: unable to open or locate the file')
else:
    count = 0               # Initialize recount cout to zero
    for line in file1:      # Read each line as text
        print(line.strip()) # Print the record
        count = count + 1   # increment record counter
    print("There are",count,"data records in this file")
    file1.close()           # Close the file after processing data
```

The last line of the output from running this program:

```
RESTART: C:/Users/lh266266.UALBANY/Code/dataRecordNumb.py
There are 61 data records in this file
>>>
```

Recall from Unit 2 when we used the `split()` method to identify each individual word in a sentence. What the method does is split or breakup a string and add the data to a list of separate 'chunks' using a defined separator. So if this line is read

```
    2017, January, Coastal Flood, Z, 79, NORTHEAST SUFFOLK
```

and we split the line into individual fields we could identify those records effected by coastal flooding during the month of August.

```
    fields = line.split(‘,’)
```

If we now print `fields` we have a list of all the fields which we can now access.

```
    >>> print(fields)
    ['2017', ' January', ' Coastal Flood', ' Z', ' 79', ' NORTHEAST
SUFFOLK']
```

Notice the extra blank space before the string “January” in the list `fields`. When trying to identify those records with the string “August” the extra space must be dealt with!

Let us complete the second program to solve the problem of identifying (and print) each of the New York State counties (zones) effected by coastal flooding during the month of August (`NYScountiesAug.py`).

```
# identify each of the New York State counties (zones) effected by coastal
flooding during the month of August
print("The following are the NYS counties (zones) effected by coastal
flooding during the month of August:")
try:
    file1 = open('NYScoastalFlooding.txt ','r') # try to find & open the
file
except:
    print('ERROR: unable to open or locate the file')
else:
    for line in file1:              # Read each line as text
        fields = line.split(",")    # split each line into a list of
'fields'
        if fields[1] == " August":    # check if the 2nd field is August
            county = fields[5]        # identify the county from the 6th
field
            print(county.strip())     # Print the record
    file1.close()           # Close the file after processing data
```

The output from running this program:

```
RESTART: C:/Users/lh266266.UALBANY/Code/NYScountiesAug.py
The following are the NYS counties (zones) effected by coastal
flooding during the month of August:
OSWEGO
NIAGARA
NORTHERN CAYUGA
JEFFERSON
ORLEANS
MONROE
WAYNE
>>>
```

To test if the program correctly identified all records effected by coastal flooding during the month of August you would need to manually go through each of the 61 records and flag the records that should be found and printed (or use an app such as Excel to do a sort and count for you).

The actual data set I downloaded had a total of 56,921 records and 51 different fields (a lot more data than what we just worked with!). A professional data scientist would also test her program using a subset of the large data set until she could be satisfied that the program was running correctly and producing the correct results.

Unit Summary

TBD

- Read and write programs using the Python IF and IF/ELIF/ELSE statements to implement a simple decision structures.
- Write simple exception handling code to catch simple Python run-time errors.
- Read and write programs using the Python FOR and WHILE statements to implement a simple loop structures.
- Construct and implement algorithms that use decision and loop structures.
- Apply basic file processing concepts and techniques for reading and writing text files in Python.

Practice Problems

1. Write a program to accept a number from the user and print whether it is positive or negative.

2. Accept three numbers from the user and print the greatest number.

3. Write a program that accepts a number in the range from 1 to 7 from the user and generates and displays the name of the weekday.

4. Write a program to input 5 numbers from the user and calculates and displays their sum and average.

5. Write a program that accepts an integer number and indicates whether it is negative, zero, or positive. Force the user to enter only valid data (repeatedly ask the user to enter only an integer number)

6. Provide the exact sequence of integers specified by each of the following range expressions.

   (a) range(5)

   (b) range(5, 10)

   (c) range(5, 20, 3)

   (d) range(20, 5, -1)

   (e) range(20, 5, -3)

   (f) range(10, 5)

   (g) range(0)

   (h) range(10, 101, 10)

   (i) range(10, -1, -1)

   (j) range(-3, 4)

   (k) range(0, 10, 1)

7. Write a program to print a pattern like a right angle triangle with numbers where each number will repeat in a row. The pattern is as follows:

```
1
22
333
4444
55555
```

8. Write a program to print a pattern like a pyramid with numbers where each number will repeat in a row. The pattern is as follows:

```
       1
      2 2
     3 3 3
    4 4 4 4
```

9. Find a poem you like and save it as a text file named poem.txt. Write a program that counts the number of lines in your program. Print out the line count and each individual line in the program (no extra blank lines!)

10. Write a program that stores the first 100 integers to a text file named `numbers.txt`. Each number should appear on a line all by itself.

11. Write a program to find the longest word in a text file.

Appendix A: Python on your Computer

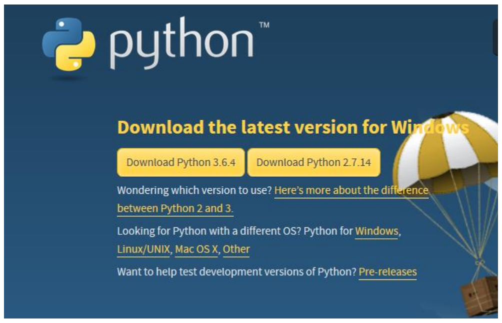
This Appendix will cover how to set up Python 3 and introduce you to IDLE, the Python development environment we will be using throughout this book.

Installing Python 3 and IDLE[23]

Installing Python is generally easy, and nowadays many Linux and UNIX distributions include a recent Python. Even some Windows computers (notably those from HP) now come with Python already installed.

With that said, first of all check that you don't already have Python installed by entering **`python`** in a command line window. If you see a response from a Python interpreter it will include a version number in its initial display. Generally any version of Python 3 will suffice.

If you need to install the Python 3 interpreter and IDLE software (IDE), you may as well download the most recent stable version. This is the one with the highest number that isn't marked as an alpha or beta release. Please see the Python downloads page, https://www.python.org/downloads/ , for the most up to date versions of Python 3. They are available via the yellow download buttons on that page. For all downloads note the directory where the files will be saved to!



Wondering which version to use? Here's more about the difference between Python 2 and 3.

Looking for Python with a different OS? Python for Windows, Linux/UNIX, Mac OS X, Other

Want to help test development versions of Python? Pre-releases

Figure 49: Python Downloads Page

- If you're running Windows: the most stable Windows downloads are available from the Python for Windows page, https://www.python.org/downloads/windows/ .
- If you're running Windows XP: a complete guide to installing *ActivePython* is at Python on XP: 7 Minutes To "Hello World!", http://dooling.com/index.php/2006/03/14/python-on-xp-7-minutes-to-hello-world/ . *ShowMeDo* (https://wiki.python.org/moin/ShowMeDo) has two videos for downloading (http://showmedo.com/videos/series?name=pythonOzsvaldPyNewbieSeries), installing and getting started with Python on a Windows XP machine - this series talks you through the Python, *ActivePython* and *SciPy* distributions.
- If you are using a Mac, see the Python for Mac OS X page, https://www.python.org/downloads/mac-osx/ . MacOS 10.2 (Jaguar), 10.3 (Panther), 10.4 (Tiger) and 10.5 (Leopard) already include various versions of Python.

Once installed you should be all set to start using the Python development environment (IDLE) on your computer.

[23] based on material from https://www.python.org/

Using Python and its IDE[24]

IDLE is Python's Integrated Development and Learning Environment (IDE) and is included as part of the Python Standard Library which is distributed with Python 3. IDLE is the standard Python development environment. Its name is an acronym of "**I**ntegrated **D**eve**L**opment **E**nvironment". It works well on both Unix and Windows platforms.

It has a Python Shell window, which gives you access to the Python interactive mode. It also has a file editor that lets you create and edit existing Python source files.

IDLE's menus dynamically change based on which window is currently selected. Each menu documented below indicates which window type it is associated with.

File menu (Shell and Editor)

New File
Create a new file editing window.

Open…
Open an existing file with an Open dialog.

Recent Files
Open a list of recent files. Click one to open it.

Open Module…
Open an existing module (searches sys.path).

Class Browser
Show functions, classes, and methods in the current Editor file in a tree structure. In the shell, open a module first.

Path Browser
Show sys.path directories, modules, functions, classes and methods in a tree structure.

Save
Save the current window to the associated file, if there is one. Windows that have been changed since being opened or last saved have a * before and after the window title. If there is no associated file, do Save As instead.

Save As…
Save the current window with a Save As dialog. The file saved becomes the new associated file for the window.

Save Copy As…
Save the current window to different file without changing the associated file.

[24] Based on material from https://docs.python.org/3/library/idle.html

Print Window
  Print the current window to the default printer.

Close
  Close the current window (ask to save if unsaved).

Exit
  Close all windows and quit IDLE (ask to save unsaved windows).

Edit menu (Shell and Editor)

Undo
  Undo the last change to the current window. A maximum of 1000 changes may be undone.

Redo
  Redo the last undone change to the current window.

Cut
  Copy selection into the system-wide clipboard; then delete the selection.

Copy
  Copy selection into the system-wide clipboard.

Paste
  Insert contents of the system-wide clipboard into the current window.

The clipboard functions are also available in context menus.

Select All
  Select the entire contents of the current window.

Find…
  Open a search dialog with many options

Find Again
  Repeat the last search, if there is one.

Find Selection
  Search for the currently selected string, if there is one.

Find in Files…
  Open a file search dialog. Put results in a new output window.

Replace…
  Open a search-and-replace dialog.

Go to Line
  Move cursor to the line number requested and make that line visible.

Show Completions
  Open a scrollable list allowing selection of keywords and attributes. See Completions in the Tips sections below.

Expand Word
Expand a prefix you have typed to match a full word in the same window; repeat to get a different expansion.

Show call tip
After an unclosed parenthesis for a function, open a small window with function parameter hints.

Show surrounding parens
Highlight the surrounding parenthesis.

Format menu (Editor window only)

Indent Region
Shift selected lines right by the indent width (default 4 spaces).

Dedent Region
Shift selected lines left by the indent width (default 4 spaces).

Comment Out Region
Insert ## in front of selected lines.

Uncomment Region
Remove leading # or ## from selected lines.

Tabify Region
Turn leading stretches of spaces into tabs. (Note: We recommend using 4 space blocks to indent Python code.)

Untabify Region
Turn all tabs into the correct number of spaces.

Toggle Tabs
Open a dialog to switch between indenting with spaces and tabs.

New Indent Width
Open a dialog to change indent width. The accepted default by the Python community is 4 spaces.

Format Paragraph
Reformat the current blank-line-delimited paragraph in comment block or multiline string or selected line in a string. All lines in the paragraph will be formatted to less than N columns, where N defaults to 72.

Strip trailing whitespace
Remove trailing space and other whitespace characters after the last non-whitespace character of a line by applying str.rstrip to each line, including lines within multiline strings.

Run menu (Editor window only)

Python Shell
Open or wake up the Python Shell window.

Check Module

Check the syntax of the module currently open in the Editor window. If the module has not been saved IDLE will either prompt the user to save or autosave, as selected in the General tab of the Idle Settings dialog. If there is a syntax error, the approximate location is indicated in the Editor window.

Run Module
Do Check Module (above). If no error, restart the shell to clean the environment, then execute the module. Output is displayed in the Shell window. Note that output requires use of print or write. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with python -i file at a command line.

Shell menu (Shell window only)

View Last Restart
Scroll the Shell window to the last shell restart.

Restart Shell
Restart the Shell to clean the environment.

Interrupt Execution
Stop a running program.

Debug menu (Shell window only)

Go to File/Line
Look on the current line. with the cursor, and the line above for a filename and line number. If found, open the file if not already open, and show the line. Use this to view source lines referenced in an exception traceback and lines found by Find in Files. Also available in the context menu of the Shell window and Output windows.

Debugger (toggle)
When activated, code entered in the Shell or run from an Editor will run under the debugger. In the Editor, breakpoints can be set with the context menu. This feature is still incomplete and somewhat experimental.

Stack Viewer
Show the stack traceback of the last exception in a tree widget, with access to locals and globals.

Auto-open Stack Viewer
Toggle automatically opening the stack viewer on an unhandled exception.

Options menu (Shell and Editor)

Configure IDLE
Open a configuration dialog and change preferences for the following: fonts, indentation, keybindings, text color themes, startup windows and size, additional help sources, and extensions (see below). On OS X, open the configuration dialog by selecting Preferences in the application menu. To use a new built-in color theme (IDLE Dark) with older IDLEs, save it as a new custom theme.

Non-default user settings are saved in a .idlerc directory in the user's home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in .idlerc.

Code Context (toggle)(Editor Window only)

Open a pane at the top of the edit window which shows the block context of the code which has scrolled above the top of the window.

Window menu (Shell and Editor)

Zoom Height
Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog.

The rest of this menu lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

Help menu (Shell and Editor)

About IDLE
Display version, copyright, license, credits, and more.

IDLE Help
Display a help file for IDLE detailing the menu options, basic editing and navigation, and other tips.

Python Docs
Access local Python documentation, if installed, or start a web browser and open docs.python.org showing the latest Python documentation.

Turtle Demo
Run the turtledemo module with example python code and turtle drawings.

Additional help sources may be added here with the Configure IDLE dialog under the General tab.

Context Menus

Open a context menu by right-clicking in a window (Control-click on OS X). Context menus have the standard clipboard functions also on the Edit menu.

Cut
Copy selection into the system-wide clipboard; then delete the selection.

Copy
Copy selection into the system-wide clipboard.

Paste
Insert contents of the system-wide clipboard into the current window.

Editor windows also have breakpoint functions. Lines with a breakpoint set are specially marked. Breakpoints only have an effect when running under the debugger. Breakpoints for a file are saved in the user's .idlerc directory.

Set Breakpoint
Set a breakpoint on the current line.

Clear Breakpoint
Clear the breakpoint on that line.

Shell and Output windows have the following.

Go to file/line
Same as in Debug menu.

Automatic indentation

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, Backspace deletes up to 4 spaces if they are there. Tab inserts spaces (in the Python Shell window one tab), number depends on Indent width. Currently, tabs are restricted to four spaces due to Tcl/Tk limitations.

See also the indent/dedent region commands in the edit menu.

Text colors

Idle defaults to black on white text, but colors text with special meanings. For the Shell, these are Shell output, Shell error, user output, and user error. For Python code, at the Shell prompt or in an editor, these are keywords, built-in class and function names, names following class and def, strings, and comments. For any text window, these are the cursor (when present), found text (when possible), and selected text.

Text coloring is done in the background, so uncolorized text is occasionally visible. To change the color scheme, use the Configure IDLE dialog Highlighting tab. The marking of debugger breakpoint lines in the editor and text in pop-ups and dialogs is not user-configurable.