MITx 6.00.1x

Introduction to Computer Science and Programming Using Python

Lecture Videos - Notes

### Table of Contents

[**WEEK 1**](#_4d0kuodhv4kd)

[Introduction](#xpmvc53wvwye)

[Python Basics](#hue50nhy1irz)

[Conditionals](#dbfvfuli1gr)

[Loops](#un9zaul91u8k)

[**WEEK 2**](#_3ztm9xjuxdcb)

[Binary System](#ktnns7exrwhp)

[Functions](#vvett0evg0mh)

[Modules](#4gro4a74r403)

[**WEEK 3**](#_xv5vg78ud8da)

[Data Structures](#i82uwyg4xzp1)

[**WEEK 4**](#_y47e15l45u5j)

[Debugging](#silg7jaufg7)

[Exception Handling](#2zao5j8mrfw5)

[**WEEK 5**](#_yp2447gd1rq7)

[Classes](#2s1n4xvbl8a9)

[Generators](#xsr6wkpxjlg7)

[**WEEK 6**](#_i47uwsa0otc5)

[Algorithmic Complexity](#ppewvdpq0c2u)

[Simple Algorithms](#z86xv2vyd84b)

[**WEEK 7**](#_e5ivw64cxshy)

[Plotting Graphs](#ejgbf1jtlcy)



## 

## **WEEK 1**

**Introduction**

Fundamentally, computers perform calculations and store results.

Although computers are fast, if we don’t give them good algorithms to work with, they won’t be very useful. They would only be slow and inefficient.

**Declarative knowledge** is statements of fact. For instance,

* There are cookies in the jar.

**Imperative knowledge** is a recipe for how to find a fact,

* Go to the kitchen.
* Stand next to the counter.
* Open up the shelf.
* Look inside the jar.

Computers oftentimes work with imperative knowledge to help us solve problems.

An **algorithm** is a set of instructions that help us achieve a result (similar to the example above of the cookies in the jar). All algorithms have:

1. A sequence of simple steps.
2. A flow of control process that specifies when each step is executed.
3. A means of determining when to stop.

A **fixed program** computer is a computer designed to only calculate a particular program (e.g. a calculator only calculates math operations).

A **stored program** computer is a computer designed to load and calculate different programs (like a normal computer can do many different things).

Computers are mostly made of three things:

1. Memory, to store both data and instructions for computing things.
2. Arithmetic Logic Unit (ALU), which does simple calculations with data from the memory.
3. Control Unit, which keeps track of what operation the ALU should be doing at a specific time (using a program counter).

Alan Turing (a great computer scientist) showed that you could compute anything you wanted using a few simple operations *(move left*, *move right*, *scan*, *print*, *erase*, and *do nothing*). He also showed that anything which could be computed in one programming language could also be computed in another.

The **interpreter** is a special program that executes a sequence of instructions in order and uses tests to figure out whether it should redo the current calculation, move on to the next instruction, or stop.

Every programming language can be thought of as:

1. A set of primitives (simple operations we can perform).
2. A means of combination (ways we can put those primitives together to form **expressions**).
3. A means of abstraction (ways to take an expression and treat it like a primitive).

The **syntax** of a programming language is the set of rules that defines how the code should be written.

* 8”hello” - incorrect syntax, because there’s no operation between 8 and “hello”
* Syntax errors are rather common and are easily caught.

**Static semantics** describes whether or not an expression with a correct syntax has meaning.

* 8 + “hello” - correct syntax, but you can’t add an integer and a string
* Static semantic errors can keep the program from running or cause unpredictable behavior.

**Semantics** describes the meaning of an expression with the correct syntax. It may be what was intended, or what was not.

* 8 + 16 - correct syntax with meaning (8 + 16 = 24)
* Semantic errors occur when the program gives a different meaning than what the programmer intended. *Garbage in, garbage out.*

**Note:** Another type of error is called a **runtime error**, which is similar to a syntax error, except that it is only caught when the program is run. Syntax errors are usually caught before a program is executed.

**Python Basics**

A program is simply a sequence of definitions (which are evaluated) and commands (which are executed by the interpreter).

**Primitives** are the fundamental objects that represent data in Python**.** All objects have a **type** that determines what kinds of things programs can do with them.

Objects can either be:

1. Scalar (cannot be subdivided into smaller parts)
2. Non-scalar (have an internal structure that can be accessed)

Scalar objects in Python include:

* int - represent integers (e.g. 5 or 12)
* float - represent real numbers (e.g. 3.14 or 7.001)
* bool - represent either True or False values
* NoneType - represent only one data type, None

You can use the type() procedure to find out the type of a value (e.g type(3)will output int).

You can also use the int() and float() procedures to cast, or convert, values into each other.

* float(3) outputs 3.0
* int(3.9) outputs 3

Expressions are combinations of objects and operators (e.g. <object> <operator> <object>).

The following are common math operators in Python:

* i+j - the sum of i and j (as a float if i or j is a float, otherwise an integer)
* i-j - the difference between i and j (same as above)
* i\*j - the product of i and j (same as above)
* i/j - the quotient of i and j (as a float)
* i//j - the quotient of i and j as an integer (remainder excluded)
* i%j - the remainder of the quotient of i and j
* i\*\*j - i to the power of j

Python follows the order of operations (everything is read from left to right in the order they appear):

1. \*\* - powers
2. () - parentheses
3. \* or / - multiplication or division
4. + or - - addition or subtraction

A **variable** is a name associated with a value. In Python, you can create variables using the following syntax:

variable\_name = value

e.g.

age = 16

animal = “buffalo”

is\_it\_raining = False

Variables are useful because we can easily access and reuse their values (as opposed to recalculating values every time we need it). They also make code easier to read and work with.

The equals sign (=) assigns the name on the left to the value on the right (which is stored in the memory). This process of assignment is called **binding** (since we’re binding a name to a value). We can retrieve the value associated with the variable by invoking its name.

You can use the += operator to increment variables:

radius = 4 (associate the value 4 with radius)

radius += 1 (add 1 to the value of radius)

You can re-bind variable names using new assignment statements:

animal = “buffalo”

animal = “flamingo”

In the above example, the old value associated with animal (“buffalo”) will be lost.

The following are common comparison operators in Python:

* i>j - greater than
* i>=j - greater than or equal to
* i<j - less than
* i<=j - less than or equal to
* i==j - equal to
* i!=j - not equal to

All of the above comparison operators will return True if true and False if false.

The following are common logic operators on booleans in Python:

* not a - will return the opposite of the true/false value that a is
* a and b - will return True only if both a and b are true
* a or b - will return True if either a or b are true

…

Another type of value is the **string**, a non-scalar value that consists of a sequence of characters such as letters, digits, or spaces.

In Python, they can be created by assigning a name to anything enclosed in single or double quotes:

name = ‘mark’

sentence = “Let’s go to the park.”

Since strings are non-scalar, they can work with a number of different operations:

‘hello’ + ‘mark’ - concatenates (or adds) strings (returns “hellomark”)

3 \* ’mark’ - successive concatenation (returns “markmarkmark”)

len(‘mark’) - outputs the length of, or number of characters in, “mark” (returns 4)

‘mark’[0] - gives the letter at the index 0 (returns “m”)

‘mark’[1:3] - gives all the letters starting at the index 1 up to, but not including, 3 (returns “ar”)

‘fantastic’[0:8:2] - gives all the letters starting at index 0 up to, but not including 8, with a step of *two* (returns “fnat” since it gives every *second* character after the first one)

str(35) - use the str() function to cast other variable types into strings (returns “35”)

Strings are immutable, which means they cannot be modified after they have been created. In order to change a string, you would have to redefine (i.e. my\_string = my\_string + “hello”)

…

An **IDE** (integrated development environment) is a combination of (usually) a *text editor* to write, edit, and save programs, a *shell* as a place to interact with and run programs, and an *integrated debugger* to debug code.

You can use the print() and input() operators to get output and input in Python:

print(“Let’s eat a cookie.”) - returns the value in the parentheses

input(“Type any letter: “) - returns the value in the parentheses while asking for input

**Conditionals**

A **branching program** is a program that can run down many different ways depending on the outcome of a test. An example of this is a conditional statement. It consists of a test, a *true block* to run if the outcome of the test is true, and (optionally) a *false block* to run if the outcome of the test is false.

In Python, you can write a conditional with the if and else operators:

if 2 < 3: - a test that will either output true or false

print(“2 is greater than 3!”) - what to do if the test outputs true

else:

print(“2 is not greater than 3.”) - what to do if the test outputs false

Each indented set of expressions in an if/else clause denotes a block of instructions.

You may nest conditionals (conditionals within conditionals):

if 2 < 3:

if 5 > 4:

print(“Yay!”)

else:

print(“Aww!”)

elif 2 < 4:

print(“Phew!”)

else:

print(“Hmm.”)

elif gives the conditional another test to check if the first one outputs false. There can be multiple elif operators in a conditional.

else gives the conditional a set of instructions if all other tests output false.

if, elif, and else are examples of **keywords** (or *reserved words*). Python uses them to recognize the structure of a program. They cannot be used as names for variables.

The control flow then for branching programs is:

if <conditional>:

<expression>

elif <conditional>:

<expression>

...

else <conditional>:

<expression>

Each conditional will output a true or false value, and the indented expressions beneath them will only run if the conditional above outputs true.

**Loops**

**Looping programs** are programs that repeat themselves until they satisfy some condition.

Using the while operator is an example of a looping program in Python. Their general format is:

while <condition>:

<expression>

* The <condition> will evaluate either a True or False value.
* If the <condition> is true, the while loop will run the <expression> below it.
* The while loop will then check the <condition> again.
* It will keep running the <expression> beneath it until the <condition> is False.

For example:

n = 1

while n < 5:

print(n)

n = n + 1

The while loop above will keep running until n is not less than five.

Another type of looping program in Python is the for loop. It’s general format is:

for <variable> in <expression>:

<expression>

For example:

for n in range(5):

print(n)

* The range() operator returns a list of values (in this case 0 to 4).
* The first time the program goes through the for loop, it will treat n as the first value in the expression (in this case, the expression is range(5)).
* The program will do whatever the bottom expression says while treating n as the first value in the expression (which is in this case 0).
* The next time the program runs through the for loop, it will treat n as the second value in the expression (it will do the same thing but treat n as, in this case, 2).
* The program will keep doing this until it has done this for all the values in the expression.

The <variable> used in for loops is simply a placeholder for the actual values the loop will be running through the expression below.

Both while and for loops can be stopped in the midst of running with the break statement:

n = 0

for x in range(10):

n = n + 1

if n == 5:

break

The above for loop will stop once n == 5 because of the break statement.

break statements immediately exit whatever loop they are in and skip the existing remainders of code.

while loops versus for loops:

* All for loops can be written as while loops, but not all while loops can be written as for loops.
* for loops have a known number of iterations, while while loops have an unknown number of iterations.
* Both while and for loops can end early from a break statement.

Use the bool() function to determine whether things are True or False:

bool(“hello world”) - returns True

bool(72) - returns True

bool(0) - returns False

The general structure of range() is:

range(start, stop, step)

* The start is the start value for the range() function (included in output).
* The stop is the end value for the range() function (not included in output).
* The step is the number of values to skip between each outputted value.

For example:

range(1, 10, 2) - will return 1, 3, 5, 7, 9

**Iteration** is the repetition of a process in order to generate a (possibly unlimited) sequence of outcomes. Each repetition of the process is a single iteration, and the outcome of each iteration is then the starting point of the next iteration. *- Wikipedia*

One characteristic of looping programs is a **loop variable**. Loop variables are:

* Initialized (created) outside of the loop.
* Changed inside the loop.
* Used in the true/false test to determine whether the loop should continue.

For example:

num = 5

while num >= 10:

print(n)

num = num + 1

The variable num in the above example is the loop variable. Loop variables are especially necessary in while loops, while for loops may use an expression to keep count of iterations.

Since strings are sequences of characters, they can be used as the expression in for loops:

for letter in my\_string:

if letter == “i” or letter == “u”:

print(“There is an ‘i’ or ‘u’!”)

**Exhaustive enumeration** is the process used to guess a value for a solution, check whether the solution is correct, and keep guessing until a solution is found or all possible values have been guessed. It is a type of algorithm that often implements looping programs.

**Bisection search** is the process used to guess a value for a solution, check whether that value is too high or too low to be correct, eliminate all other values that may be too high or too low as well, and keep guessing from the remaining values until a solution is found or all possible values have been guessed. It is a type of algorithm that often implements looping programs as well.



## **WEEK 2**

**Binary System**

Computers calculate everything using the **binary system**, a number system that uses a base of two instead of the more common base of ten. Just like numbers can be represented by the addition of powers of ten in the base ten number system, numbers can be represented by the addition of powers of two in the base two number system:

357 = (3 \* 102) + (5 \* 101) + (7 \* 100) - base 10 number system

357 = 28 + 07 + 26 + 25 + 04 + 03 + 22 + 01 + 20 - base 2 number system

In binary, the number shown above (357) would be written 101100101. 1s indicate the power of two is “on,” or counted, while 0s indicate the power of two is “off,” or not counted (like, 357 is 256 + 64 + 32 + 4 + 1).

Just like in the base ten number system, where numbers divided by ten have all of their digits shifted to the right by one, numbers divided by two in the base two number system do the same thing as well:

4326 // 10 = 432 - 4326 divided by 10 is 432 (ignoring the remainder)

10011 // 2 = 1001 - in binary, 10011 divided by 2 is 1001 (ignoring the remainder)

To represent decimals or fractions in binary, multiply the desired decimal or fraction by a power of two that will make it a whole number. Convert that whole number into binary form and then divide the binary form by the same power of two used to make the decimal or fraction a whole number:

⅜ = 0.375 = (3 \* 10-1) + (7 \* 10-2) + (5 \* 10-3)

0.375 \* 23 = 3 - multiply the decimal by 23 to make it a whole number (3)

3 = 11 - convert 3 to binary (11)

11/23 = 0.011 - divide 11 by 23 (simply shift all the digits to the right three times)

Some implications of representing fractions in binary include:

* If there is no power of two that when multiplied by the fraction will make it a whole number, the binary representation of that fraction will always be an approximation.
* Because of that, it is better to avoid comparing two floats to each other with a “==,” since their binary representations may not always be exactly the same. It’s better to compare fractions by abs(fraction\_A - fraction\_B) <= some\_small\_number or some other alternative method.

**Functions**

* In programming, **abstraction** refers to the idea not necessarily knowing (exactly) how something works but knowing how to properly use it instead. For example, you may not know what a module of code consists of or how it was built, but you can still use it to achieve your intended results if you know how to use it and what to use it for.
* **Decomposition** refers to the idea of breaking a problem down into smaller, self-contained pieces. For example, you may divide a very large code project into smaller, individual pieces of code that you can later combine to achieve a finished project.

Both abstraction and decomposition are important ideas that help simplify things when working with code that can easily get complicated and messy (such as in large projects).

**Functions** are one way to implement the ideas of abstraction and decomposition in programming. Functions are reusable pieces of code that are given a name and perform calculations with the arguments they are given.

In Python, functions are written with the keyword def followed by the function name and any arguments it may have in parentheses (multiple arguments are separated by commas):

def function\_name(argument 1, argument 2):

<expression>

The first line of a function ends with a colon and the following lines of a function (which consist of expressions) are indented. Arguments are the values given to functions to work with. Inside functions, arguments are known as **parameters**. Parameters serve as placeholder variables for the function to perform calculations with:

def add\_this(number\_1, number\_2):

total = number\_1 + number\_2

return(total)

Functions are called (or **invoked**) by simply typing the function name and including any parameters it uses:

add\_this(5, 8)

The above function will treat the argument 5 as the parameter number\_1 and the argument 8 as the parameter number\_2. Since the expressions in add\_this add those two numbers and print the result, the example above will return the value 13.

The return function returns the value it is given (in this case, from a function). It differs from print() in that print() displays values while return returns values that can be used. For example:

def add\_this(number\_1, number\_2):

total = number\_1 + number\_2

print(total)

some\_variable = add\_this(3, 8)

When the function add\_this is called (when some\_variable is defined), it will print 13 to the console but return the value None to some\_variable (functions will return None if they don’t contain a return() statement). some\_variable will then be equal to None even though add\_this added 3 and 8 printed 13.

def add\_this(number\_1, number\_2):

total = number\_1 + number\_2

return total

In this above example, however, when the function add\_this is called (when some\_variable is defined), it will not print any value but return the value 13 to some\_variable. some\_variable will then be equal to 13 even though add\_this didn’t print anything.

* Unlike print(), return doesn’t require parentheses when called.

Also, the return statement can be used to return multiple values (by separating each of the additional return values with a comma):

return num, total, result

A function’s **scope** describes the range of a function (what it can or can’t do). Any variables or parameters that are created inside a function are **local**, which means they only exist inside the function. Once the function ends, those variables and parameters are destroyed, and cannot be called in the **global scope** (the world outside of a function, where regular variables are created). Because of this, functions can access variables outside (in the global scope), but cannot modify those variables in the global scope.

x = 10

def my\_function(arg\_1):

x = 3

print(x + arg\_1)

return x

In the above example, x = 10 is defined in the global frame. The function my\_function creates a local variable also named x, but equal to 3 instead. my\_function prints the sum of the local variable x (which is 3) added to the parameter arg\_1, and then returns the value of the local variable x (which is 3). Once my\_function is finished running, calling the variable x will invoke the variable x from the global frame (which is 10). The local variable x that was equal to 3 only existed within my\_function and was destroyed once my\_function ended.

x = 10

def my\_function(arg\_1):

print(x)

The above function will print 10 if called, since no local value of x is defined and therefore my\_function accesses the global variable x.

x = 10

def my\_function(arg\_1):

x = x + 5

The above function cannot access the value of x in the global frame, because if so would be changing the value of the global variable x—something beyond the scope of a function. Since there has been no local variable named x that has been defined within the function, the above function will output an error.

Functions can return other functions. In such cases, the parameters for the function being returned must be included when the function is called:

my\_function(arg1, arg2)(arg1, arg2, arg3)

In the above example, the second set of parentheses contains the parameters of whatever function my\_function returns (and the returned function will take those values and output another value).

When defining functions, the arguments that a function uses can be given default values by simply assigning values to them when they are defined:

def my\_function(arg1, arg2, arg3 = True):

<expression>

A function like the one above can accept only two arguments if necessary, but will override the default value of arg3 if it is given an additional argument.

To make code easier to read and understand, functions traditionally include a **docstring** that specifies the assumptions (the conditions that the function assumes are met in order to run) and guarantees (the conditions that the function will meet if all of its assumptions are met) of the function. Docstrings are written as multi-line comments, which are written inside two sets of three double quotes:

def my\_function(arg1, arg2):

“””

This is the docstring.

It specifies the assumptions of this function…

and the guarantees of it, too!

“””

<expression>

Giving the built-in function help() any function name as an argument (e.g. help(my\_function)) will display the name, arguments, and docstring of that function.

**Recursion**

**Recursive** functions are functions that call themselves. Recursion is often used in ways similar to iteration, and many calculations can often be computed using either process. However, iteration and recursion have their own pros and cons that can vary depending on the calculation being computed.

def factorial(number):

if number == 0 or number == 1:

return

else:

return number \* factorial(number-1)

The above function uses recursion to calculate the factorial of a given number. The function factorial multiplies the argument number by the result of itself called on the difference of one less than the number until the argument number is equal to 1 or 0. Because of this, the else clause in the function will keep running until number is equal to 0 or 1. Then, number (which will usually be equal to 1) will be returned to the factorial function that just called it (line 5), which will multiply it by number and return the product to the factorial function that called it (line 9), and keep repeating this process until all the factorial functions that have been called are satisfied.

Functions often contain a **base case** to avoid ending up in an infinite loop from calling itself too many times. A base case ensures that at some point, some value will be returned, and because of that, no more function calls will be made and all the functions that have been called will begin to be satisfied. In the above example, the base case

if number == 0 or number == 1:

return number

ensured that the function factorial would begin looping back at some point (because when number was equal to 1 or 0 some value (usually 1) was returned and the loop will not continue endlessly.

* So in essence, recursion systematically reduces a problem (into a smaller or simpler version of that problem) until it can be solved directly and then successively returns the calculated values to the functions that called for them (in this case, factorial reduced the problem to a series of simple multiplications) until a final solution is returned.

Recursive functions can contain multiple base cases and more than one instance of the function in the recursive call (as seen in the following function):

def fibonacci(x):

if x == 0 or x == 1:

return 1

else:

return fibonacci(x-1) + fibonacci(x-2)

**Modules**

**Modules** are Python files that contain a collection of variables and functions. They are often used to make larger projects easier to maintain by dividing them into smaller pieces or simply to group a bunch of related variables and functions together in a single file.

To use the variables and functions in a module in another Python file (or even in the shell), use the import statement followed by the name of the desired module:

import math

To call the variables or functions from an imported module, use the name of the module followed by a period and the name of the desired variable of function:

math.pi

math.log(100)

To avoid using **dot notation** (that clunky module.name syntax) when using items from modules, use the from keyword to import the desired variables or functions:

from math import pi

from math import log

To import all the variables and functions from a module, use \* (which is a placeholder for *all*):

from math import \*

When using the above method of importing modules, be sure that none of the names of variables or functions from the module collide with any names of variables and functions in the file.

If you wish to import a module, but refer to it by a different name than it is given in your code, use the as keyword:

import math as funky

The name that appears after the keyword as is the name that the imported module will be referred to by. In the above example, after importing the module math as funky, the call math.pi will throw an error, but the call funky.pi will be perfectly fine.

If you want to open up other files to either read or write information to them, use the open() command in Python:

my\_file = open(“some\_file”, “w”)

The first argument open() takes is the name of the file you want to open and the second argument indicates what you want to do with that file (in this case, “w” means the file is opened to write information to it). In the above example, the variable my\_file is assigned to a file named some\_file.

You can use the .write() function to write information to a file or the .close() function to close a file when you’re finished:

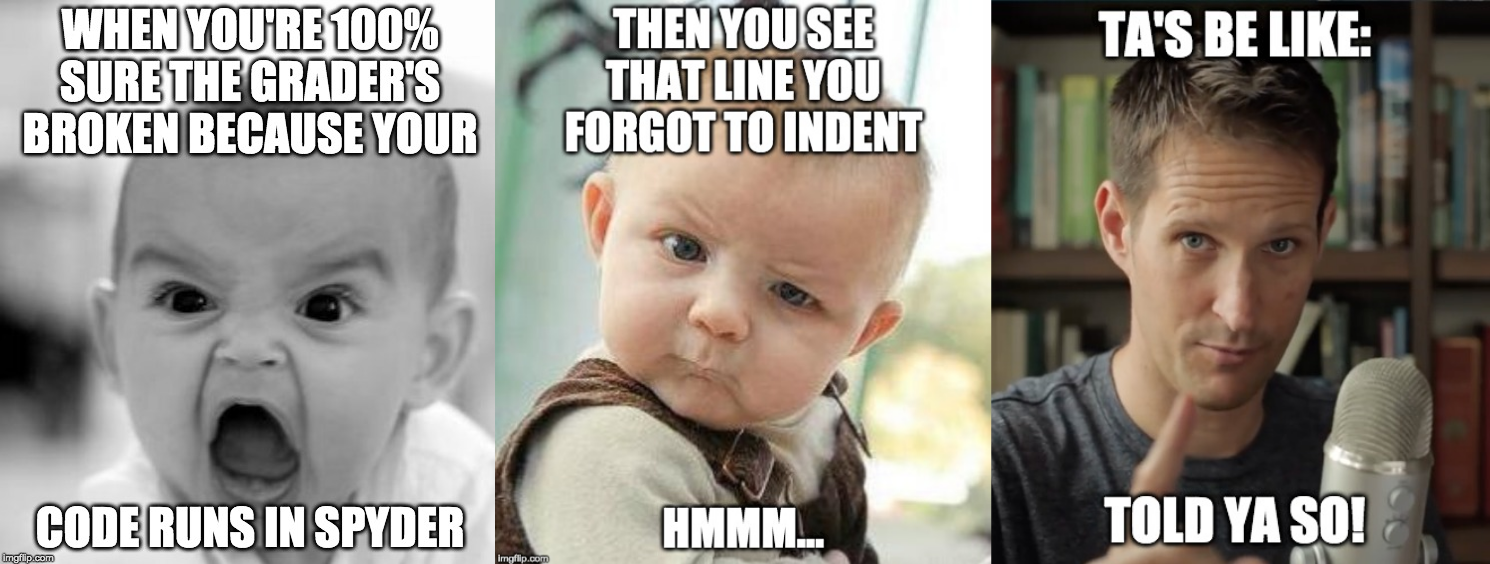
my\_file.write(“Hello, there!” + “\”)

my\_file.close()

You can open a file to read (by using “r” instead of “w” as the second argument) to read information in the file:

for line in my\_file:

print(line)



## **WEEK 3**

**Data Structures**

**Tuples** are a type of data structure in Python that consist of an ordered sequence of elements. A single tuple can contain different types of elements (such as integers, strings, or even other tuples). Tuples are **immutable**, which means like strings, values within tuples can be accessed but not modified.

In Python, tuples are created by assigning a tuple name to a series of items (separated by commas) enclosed in parentheses:

my\_tuple = (“buffalo”, 8, 3.6, “hello”)

Elements in tuples can be accessed by indexing or sliced by… slicing:

my\_tuple[0] - returns “buffalo”

my\_tuple[1:3] - returns 8, 3.6

To create a tuple with a single element, include a comma after the element (to avoid confusing the parentheses with order of operations or something):

other\_tuple = (“flamingo”,)

Or to create an empty tuple:

empty\_tuple = ()

Tuples are iterable (just like strings):

for x in my\_tuple:

print(x)

Tuples can be assigned to each other:

other\_tuple = my\_tuple

(x, y) = (y, x)

Use the following syntax to access elements within tuples (or characters within strings) within tuples:

a\_tuple = (2, 3, “hello”, (9, “eight”, “hi”), 17.2)

a\_tuple[3][0] - returns the element at position 0 in the element at position 3

The above case would return 9.

**Lists** are another type of data structure in Python that also consists of an ordered sequence of elements and can be internally accessed by indexing. The difference between lists and tuples is that lists are **mutable**, which means the elements inside them can be modified. A single list can contain multiple object types (although this is often uncommon) such as strings, integers, floats, or even other lists or functions.

In Python, lists are created by assigning a list name to a series of items (separated by commas) enclosed in brackets:

my\_list = [“flamingo”, 18, “bye”, 2, 1]

Like tuples, lists are iterable and can be assigned to each other. Individual elements of a list can be modified using the following syntax:

my\_list[2] = “see you tomorrow”

The above example changes in the element at position 2 (“bye”) in the list my\_list to “see you tomorrow”.

Both tuples and lists can concatenated:

new\_tuple = my\_tuple + other\_tuple

new\_list = my\_list + some\_list

And multiplied:

my\_tuple \* 3

my\_list \* 8

The keywords in and not can be used to check for whether certain elements are in a tuple or list:

“Hello” in my\_tuple # will return ‘True’ if so (otherwise, ‘False’)

3 not in my\_list # will return ‘True’ if so (otherwise, ‘False’)

The len() function is useful for getting the *length* of a string, tuple, or list:

len(“wonderful”) - returns 9

len(my\_tuple) - returns 4 (from the above example)

The .append() function is useful for adding elements to the end of a list:

my\_list.append(“happier”)

The above example adds the elements “happier” to the list my\_list.

The .extend() function is useful for adding lists or tuples (or groups of elements) to lists:

my\_list.extend((2, 3))

The above example adds two elements, 2 and 3, to the list my\_list.

The .insert() function is useful for adding an element in a specific index in a list:

my\_list.insert(2, “mountain”)

The above example inserts the string “mountain” to the index 2 in the list my\_list.

The .remove() function is useful for removing specific elements from a list:

my\_list.remove(“flamingo”)

The above example removes the first instance of the string “flamingo” in the list my\_list.

The del function is useful for removing elements at certain indexes in a list:

del(my\_list[2])

The above example removes the element at position 2 in the list my\_list.

The .pop() function is useful for simultaneously removing and returning the last element of a list:

my\_list.pop()

To pop an element at a specific place in a list, include the index of the desired element in parentheses of .pop():

this\_list.pop(5)

The above example removes and returns the element at index 5 of the list this\_list.

The list() function is useful for converting a string into a list (with each character of the string a separate item of the list).

list(“wonderful”)

The above example would return [“w”, “o”, “n”, “d”, “e”, “r”, “f”, “u”, “l”].

The .split() function is similar to the list() function in that it converts string into a list except that it splits the characters in the string to make the elements in the list based on the character it is given:

s = “wonderful”

s.split(“e”)

The above example would split the string s into individual elements in a list whenever the character “e” is found (i.e. returning [“wond”, “rful”].

The “”.join() function is useful for joining elements in lists into strings:

another\_list = [“hey”, “what”, “wow”]

“”.join(another\_list)

The above example returns “heywhatwow”.

To insert characters between each element of a list when using the “”.join() function, simply include the desired characters in parentheses:

“\_”.join(another\_list)

The above example returns “hey\_what\_wow”.

The sorted() function is useful for returning a sorted version of a list without changing the list itself:

sorted(my\_list)

The .sort() is the same as the sorted() function except that it actually changes the list itself:

my\_list.sort()

The reverse() function is useful for reversing the order of the elements in a list:

my\_list.reverse()

Use the .count() function to count the number of elements in a list:

this\_list.count(4)

The above example returns the number of times 4 occurs as an element in this\_list.

The .index() function is useful for returning the index of an element in a list:

this\_list.index(“hello”)

The above example returns the index of the first instance of the element “hello” in the list this\_list.

**Note:** when called, the range() function actually just returns (or, more technically, yields) a tuple of numbers.

**Aliasing** is the term used to describe multiple variables that point to the same list:

colors = [“orange”, “green”, “purple”]

more\_colors = colors

In the above example, colors and more\_colors are aliases of each other since they both point to the same list. Changing this list using one variable (say, more\_colors) will change the same list accessed by the other variable (in this case colors), which can occasionally complicate things.

To avoid aliasing, **clone** lists instead of assign other variable names to them:

other\_colors = colors[:]

some\_colors = [“orange”, “green”, “purple”]

Use the == comparison operator to check whether lists are equal to each other:

colors == some\_colors # will return ‘True’

Use the is keyword to check whether two variable names point to the same list:

colors is some\_colors # will return ‘False’

In the above example, unique lists are created and there are no variables pointing to the same list.

* When iterating over lists, it is usually a good practice to avoid mutating the list while still in the process of iteration, since adding or removing elements to a list changes the indexes of the elements in the list and can potentially lead to problems such as counting errors.
* One method of avoiding this problem is by cloning a list and iterating over the cloned list while changing the original (ensuring that the list being used by the iteration does not change while the list not being used by the iteration gets mutated).

The map() function behaves like the range() function in that it yields an iterable set of values. map() takes a function and applies it to every element in one or more lists:

L1 = [1, -2, 5, -3, -4, 8, -7, 11]

for x in map(abs, L1):

print(x)

The above example would print every element of the list L1 with the abs() function applied to it (returning 1, 2, 5, 3, 4, 8, 7, and 11).

(The abs() function returns the absolute value of an integer or float (i.e. abs(-3)).)

Lists can be indexed using indexes from other lists. See the following example:

a = [1, 2, 3, 4, 0]

b = [3, 0, 2, 4, 1]

c = [3, 2, 4, 1, 5]

a[a[1]]

a[b[2]]

a[c[a[b[0]]]]

a[c[a[b[3]]]]

The above indexes on the lists a, b, and c would be equal to 3, 3, (the third index would actually end up as an IndexError), and 4, respectively.

…

**Dictionaries** are another type of data structure found in Python. They are similar to lists in that they are mutable and can handle different types of elements. They are different, however, in that store pairs of data (or in other words, allow customizable indexes).

Dictionaries are created using the following syntax:

dictionary\_name = {key: value, key: value, … }

my\_dictionary = {“Mark”: “A”, “Linda”: “B”, “Aaron”: “A+”}

Access values stored in dictionaries by using the key associated with that value:

dictionary\_name[key] - returns the value associated with the given key

my\_dictionary[“Linda”] - returns “B”

Add values to a dictionary using the following syntax:

dictionary\_name[key] = value

my\_dictionary[“Stacy”] = “A” - adds entry to the end of my\_dictionary

Check if values are in a dictionary using the following syntax:

key in dictionary\_name - returns a boolean value

“Mark” in my\_dictionary - returns True

Use the del keyword to delete entries in dictionary:

del dictionary\_name[key] - deletes entry at associated key

del my\_dictionary[“Stacy”]

Use the .keys() and .values() functions to return a tuple of all the keys or values of a dictionary:

my\_dictionary.keys()

my\_dictionary.values()

Use the .get() function to “safely” access the values of keys in a dictionary. If in the event the desired key is not in the dictionary, the .get() operator will not throw an error but rather return its second argument:

my\_dictionary.get(“a”, “Key not found!”)

In dictionaries, values can be both mutable and immutable, ranging from anything such as integers to strings to functions and even other dictionaries. Values can also be duplicates.

Use the .copy() function to return a copy of a dictionary:

another\_dictionary = my\_dictionary.copy()

In dictionaries, keys have to be unique and can only be immutable (more accurately *hashable*) types such as strings or tuples. A single dictionary can contain various types of keys in any particular order.

The global keyword is useful for creating variables within functions that can be accessed outside the scope of those functions:

def a\_function:

global count

count = 0

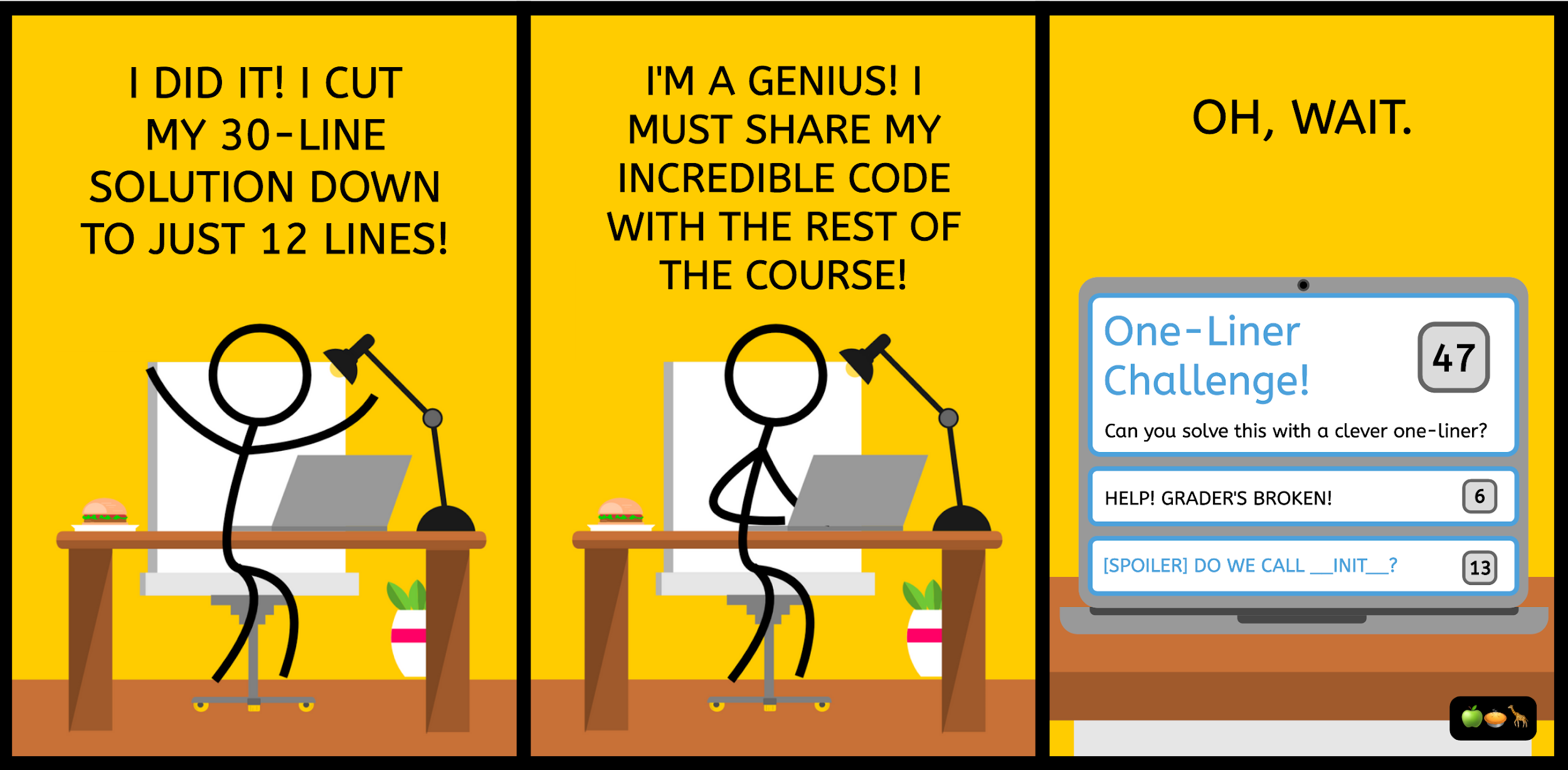
for x in range(10):

count += 1

print(count)

In the above example, the variable count can be accessed out of the function a\_function since it was declared as a global variable.

Although global variables are helpful in some instances, they are generally avoided unless they are necessary or particularly helpful because they can easily complicate things.



## 

## **WEEK 4**

**Debugging**

**Debugging** is the process of finding and removing mistakes (or, “bugs”) in code. There are various ways to go about debugging, such as by studying the events that led up to an error, purposely attempting to “break” a program in order to identify potential weaknesses, or by simply practicing **defensive programming** to make debugging easier in the first place.

*Defensive programming* is writing code in such a way as to make debugging easier and less complicated. Common practices in defensive programming include including docstrings when writing functions, breaking down projects into modules, and specifying the expected inputs, outputs, and results of modules.

There are several different types of testing when looking for bugs in code:

* **Unit testing** is making sure each component of a program works correctly by individually testing each function or module of a project.
* **Regression testing** is rechecking a piece of code after fixing a bug to ensure that the bug has been fixed and no additional bugs have been introduced in the process.
* **Integration testing** is making sure the entire program works as a whole by testing the entire program (as opposed to unit testing).
* **Black box testing** is an approach to testing that doesn’t require knowledge of how the function being tested was written. Rather, it is checking the function using different test cases based on the expected input(s), output(s), and result(s) of the function (natural partitions).
* **Glass box testing** is an approach to testing that checks a function by using different test cases based on how the function was written. It attempts to use as many test cases as necessary to check every potential path of the function (such as by exercising each branch of a conditional or running multiple variations of a loop in a function). Testing every potential path of a function may not always be possible, however, such as is the case with recursive functions, which may have a wide range of potential paths.

**Natural partitions** are test cases that intuitively check a function for different test cases based on what the function does. For instance, the natural partitions for a function that calculates the absolute values of integers may be: 0, 1, n < 0, n > 0 (which are test cases designed to check the functionality of a function).

If no natural partitions are known, **random testing** (simply providing random inputs to check a function) may be used instead.

There are several different types of bugs in programming:

* **Overt bugs** are bugs that have obvious manifestations, such as in code that crashes or runs forever. They are often relatively easy to spot.
* **Covert bugs** are bugs that don’t have obvious manifestations, such as in a value being returned by a program, but that value sometimes (or always) being incorrect. They can be more difficult to find than overt bugs.
* **Persistent bugs** appear every time code is run.
* **Intermittent bugs** appear sometimes when code is run, even when the inputs of the code are the same. They can be difficult to find but are catchable if the exact same conditions used to run the code are reproduced.

Some tips for finding bugs include:

* Use print() statements throughout a function to print out arguments, values, or results to understand what’s happening at every step of execution and pinpoint where your program may be going wrong.
* Use the error messages in the console to figure what type(s) of error(s) have occurred (i.e. TypeError(s), IndexError(s), NameError(s), etc.) and where they are located.
* In order to understand your own thought process and what’s going on in your code, imagine you are explaining the code to someone else.
* Rather than asking yourself, “What’s wrong with my code?” ask yourself, “How did I get this result?”.
* Save backup versions of your code before making changes while debugging to avoid messing things up and not being able to get back to your original code.

**Exception Handling**

**Exception** (or error) **handling** is the process of instructing a computer in what to do when an error occurs. By default, Python displays an error message and stops running when an error occurs, but this default behavior can be changed using a few keywords.

In Python, you can handle exceptions with the keywords try and except:

try:

num = int(input(“Give me a number: “))

print(“Your number was “ + str(num))

except:

print(“Oh no! An error has occurred!”)

Everything within the body of the try statement is executed until an error occurs, at which point Python skips down and executes the except clause instead of displaying an error message and terminating the program. If no error occurs, the try clause is run, the except clause is skipped, and the program runs as normal.

To include specific instructions for different cases or types of errors, use the except operator followed by the name of the error you want to handle (i.e. ValueError):

try:

num = int(input(“Enter a number: “))

print(“Your number was “ + str(num))

print(“Half of that is “ + str(num/2))

except ValueError:

print(“Sorry! A value error has occurred!”)

except TypeError:

print(“Sorry! A type error has occurred!”)

except:

print(“Oh no! An error has occurred!”)

Each of the except clauses are run if the error they handle occurs. If any other error occurs, the last except clause (unspecified) is run. If no error(s) occur, the try clause is run and all the other except clauses are skipped.

There are several common types of errors in Python:

* SyntaxError: this type of error occurs when Python cannot parse (read) a program due to errors in the program’s syntax.
* NameError: this type of error occurs when the local or global name of an object cannot be found
* AttributeError: this type of error occurs when a reference to an attribute fails
* TypeError: this type of error occurs when an operator isn’t given a correct operand
* ValueError: this type of error occurs when an operator is given the correct operand, but the value of the operand is inappropriate
* IOError: this type of error occurs when the input/output system reports an error (such as being unable to locate or open a file)

The keyword else can be used to execute code after a try clause runs with no errors:

try:

greeting = input(“Enter a greeting: “)

print(greeting)

except:

print(“Oops! An error has occurred.”)

else:

print(“Greeting successful!”)

The finally clause executes code at the end of a try statement, regardless of whether or not any errors have occurred or break, continue, or return statements have been executed within the try statement:

try:

greeting = input(“Enter a greeting: “)

print(greeting)

except:

print(“Oops! An error has occurred.”)

else:

print(“Greeting successful!”)

finally:

print(“Greeting or not, welcome to Cloud Cuckoo Land!”)

The finally clause is useful for executing instructions that you wish to be run no matter what happens.

Use the raise operator to raise exceptions (including custom exceptions) on your own:

try:

num = int(input(“Enter a number: “))

print(“Your number was “ + str(num))

print(“Half of that is “ + str(num/2))

except ValueError:

print(“Sorry! A value error has occurred!”)

except:

raise TypeError(“Oh no! A type error has occurred!”)

The raise operator raises whatever error is typed after it (in this case, TypeError) plus a custom display message (in this case, “Oh no! A type error has occurred!”).

All these exception handling operators can be used (and are quite useful) in functions.

…

The assert operator is another exception handling operator that exemplifies defensive programming quite well. The assert operator executes code and terminates a function if specified assumptions are not met.

def divide\_data(some\_data):

assert len(some\_data) > 0, “Oh no! No data received!”

return some\_data / data

In the above example, the assert statement “assumes” that the length of some\_data is longer than 0; if in the event this is not the case, the assert statement will raise an AssertionError and output, “Oh no! No data received!”, terminating the function (and therefore skipping the return statement).

Although they can be used elsewhere, assert statements are usually used to check the inputs of functions. They are useful for catching bugs and checking values for legality.



## **WEEK 5**

**Classes**

*“Python is an [****object-oriented programming language****]. Object-oriented programming (OOP) focuses on creating reusable patterns of code, in contrast to procedural programming, which focuses on explicit sequenced instructions. When working on complex programs in particular, object-oriented programming lets you reuse code and write code that is more readable, which in turn makes it more maintainable.”*

* *How To Code in Python 3 by Lisa Tagliaferri*

Object-oriented programming makes code easier to read, organize, and manage by increasing modularity in programming.

As we’ve seen already, Python supports several different types of data. Each instance of these types of data is known as an object and possesses a type (such as an integer, float, string, etc.), an internal representation in the computer (the means by which they are stored in the computer’s memory), and a set of procedures for interaction with object (such as operators or methods). Objects can be created and destroyed in Python.

Besides the types of objects already built-in to Python, you can create your own object types by using **classes**. After creating a class by defining its name and attributes, you can use that class by creating *instances* of objects from the class and performing operations on them. Classes make it easy to reuse and organize code by grouping objects that share both common attributes and procedures that operate on them.

To create a class in Python, use the class keyword followed by the name of your class and the name of the parent class in parentheses followed by a colon. By convention (tradition), class names are always capitalized in Python:

class Animal(object):

<body>

In the above example, the class object is used as the parent class to establish Animal as an object in Python and inherit all of the parent class object’s attributes. object is known as a superclass of Animal (more on this later).

The body of a class contains the **attributes** of the class. Data attributes are objects that make up the class. Procedural attributes (called **methods**) are functions that are designed specifically to work on objects of the class. The first function (or method) in a class is a special function (called \_\_init\_\_) that specifies how to create a new instance of the class:

class Animal(object):

def \_\_init\_\_(self, size, color, mood):

self.size = size

self.color = color

self.mood = mood

self.alive = True

The first parameter of the function \_\_int\_\_ is (by convention) always called self. It serves as a way to reference an instance of the class when an instance is created. The following parameters (in this case, size, color, and mood) are data attributes of every instance of the class that will be created (e.g. when an instance of the class is created, it will possess its own version of each one of these attributes).

The body of the function \_\_init\_\_ usually consists of statements that bind the parameters passed to \_\_init\_\_ (in this case, any values passed for size, color, and mood) to the instance of the class when an instance of the class is created. The dot notation locally binds a particular attribute of the instance of the class with a desired attribute. As seen in the above example, not all attributes need to be bound to a parameter from \_\_init\_\_ (as is the case with the attribute alive being passed a default value of True).

To create a new instance of a class, type the name of the instance followed by the assignment operator and the name of the class of which you want your instance to be part of. After the name of the class, provide values for all the parameters in the \_\_init\_\_ function of that class (in parentheses).

hippo = Animals(“large”, “purple”, “amused”)

There is no need for the parameter self to be passed a value because it automatically references the instance being created (in this case, the parameter self would refer to the instance hippo).

You can access particular attributes of an instance of a class by using dot notation:

hippo.size

The above example would return “large” because the \_\_init\_\_ function bound the parameter size to the attribute size for the instance hippo. The statement self.size = size in \_\_init\_\_ would have become hippo.size = “large” when the instance hippo was created.

Whenever a new instance of a class is created, a new scope, or **frame**, is created (similar to the way functions have their own scope) in which all the attributes of that new instance are bound. When the call hippo.size is made (as in the above example), Python looks up the frame hippo and finds the value associated with size (in this case, “large”) in it. In this way, there are no errors when multiple instances of a class are created and multiple attributes with the same name created as well.

Additional procedures, or methods, can be created to work the attributes of instances of a class. Methods are limited to the scope of the class in which they are created, thereby also allowing multiple methods with the same names to be created and used in different classes. By default, Python automatically passes a value for the parameter self every time a method call is made, which is why self is often the first parameter of every method of a class:

class Animal(object):

def \_\_init\_\_(self, size, color, mood):

self.size = size

self.color = color

self.mood = mood

self.alive = True

def feeling(self):

return “The”, self.color, str(self), “is feeling”, self.mood,

“.”

def colors(self, other):

return “The”, str(self), “is”, self.color, “and the”,

str(other), “is”, other.color, “.”

In the above example, the function feeling uses dot notation to access the attributes of whatever instance is passed for the parameter self. Method calls are made using dot notation as well:

print(hippo.feeling())

A method call is made by typing the name of the instance followed by a dot and the name of the desired method along with any additional arguments (in parentheses). In the above example, the method feeling only takes one argument, self, which is automatically passed by Python so no additional arguments are needed. The above example would print, “The purple hippo is feeling amused.”.

Alternatively, methods can be called by typing the name of the class followed by a dot and the name of the method will *all* the arguments it takes (in parentheses):

print(Animals.feeling(hippo))

However, using this method will require you to provide an argument for self since the method being called will need to know what instance to work on.

giraffe = Animals(“large”, “yellow”, “mellow”)

print(hippo.colors(giraffe))

The above example exemplifies a method call with an additional argument and would print, “The hippo is purple and the giraffe is yellow.”.

There are several (in fact, many) built-in methods that Python uses when function calls are made on instances of classes. For example, when called on an instance of a class, the print() function uses a method named \_\_str\_\_:

print(hippo)

The above call would print something such as <\_\_main\_\_.Animals object at 0x7fa918510488>. To change the way print() behaves on instances of the class Animal, we can redefine \_\_str\_\_ within Animal to do what we want:

class Animal(object):

def \_\_init\_\_(self, size, color, mood):

self.size = size

self.color = color

self.mood = mood

self.alive = True

def feeling(self):

return “The”, self.color, str(self), “is feeling”, self.mood,

“.”

def colors(self, other):

return “The”, str(self), “is”, self.color, “and the”,

str(other), “is”, other.color, “.”

def \_\_str\_\_(self):

return “< A”, str(self), “is an animal! >”

Calling the print() function on an instance of the class Animal (say, hippo) will now print:

< A hippo is animal! >

Other built-in methods which can be redefined include the \_\_add\_\_ and \_\_eq\_\_ methods, which change the way the + and == operators work, respectively. For example, adding the following code to the end our Animal function will change the way expressions such as hippo + giraffe or hippo == giraffe are run:

def \_\_add\_\_(self, other):

return “What do you get when you cross a”, str(self), “with

a”, str(other) + “?”

def \_\_eq\_\_(self, other):

return “What’s the difference between a”, str(self), “and

a(n)”, str(other) + “?”

When creating classes, it is helpful to remember what objects consist of. The name of the class specifies the *type* of the object being created, the \_\_init\_\_ function specifies the way objects of that class are *internally represented*, and the rest of the functions specify the way objects of that type can be *interacted* with. Classes are *types* while their instances are *objects* of those types. Typically, the functions in a class are operations specially designed to meet the needs and specifications of objects of that class.

When creating classes, it is generally a good practice to avoid directly manipulating attributes of an instance of that class (such as by using dot notation to directly access and change attributes (e.g. hippo.color = “purplish-brown”)). Instead, it is better to include separate functions in the class that return the values of certain attributes (known as **getters**) and other functions that change the values of certain attributes (known as **setters**). This is to ensure that there is no confusion when accessing or changing the values of attributes (especially in large, complex classes where multiple values may depend on each other) and to make things less complicated for other people who may use your class (e.g. other programmers need not know how you created your class, only how to use the functions within it (a form of abstraction)).

class Rectangle(object):

def \_\_init\_\_(self, length, width):

self.l = length

self.w = width

self.area = length \* width

def getLength(self):

return self.l

def getWidth(self):

return self.w

def getArea(self):

return self.area

def setLength(self, newLength):

self.l = newLength

self.area = self.l \* self.w

def setWidth(self, newWidth):

self.w = newWidth

self.area = self.l \* self.w

In the above example, the various functions after \_\_init\_\_ ensure that the attributes of any instance of Rectangle can be accessed and updated efficiently. In the case of setLength and setWidth, the attribute self.area is updated appropriately along with newLength or newWidth.

Classes can be used to create **hierarchies**, or groups of **parent classes** (also called superclasses) and **child classes** (also called subclasses). A subclass is a class that inherits all of the attributes and methods of its associated superclass. It may contain new attributes or methods, use existing attributes or methods from its parent class, or redefine existing attributes or methods from its parent class to suit its own needs.

The name (in parentheses) after the name of a class is the name of the parent class from which that class inherits. In most cases, the first class in a hierarchy is set to inherit from the class object, a built-in class from Python which contains a number of basic methods (such as \_\_init\_\_ and \_\_str\_\_). The rest of the classes in a hierarchy typically inherit from each other:

class Dog(Animal):

def sound(self):

return “Bark!”

def action(self):

return “The dog rolled over.”

def \_\_str\_\_(self):

return “< A”, str(self), “is a dog! >”

In the above example, the class Dog is created as a subclass of the class Animal. There is no \_\_init\_\_ method because the class Dog can inherit that method from the class Animal, but there are two new methods (sound and action) and an existing method from the class Animal (\_\_str\_\_). The two new methods can be used on any instance of the class Dog, and so can any methods included in the class Animal (such as feeling or colors). The \_\_str\_\_ method, being redefined, will now only return “A <self> is a dog!” if called on an instance of the class dog. To access the original \_\_str\_\_ method from the class Animal, use the following syntax:

Animal.\_\_str\_\_(<instance>)

The dot notation specifies which method from which class is to be used, and <instance> is the name of the instance you want the method to be called on.

An instance of the class Dog can now be created:

retriever = Dog(“medium”, “golden-brown”, “excited”)

And worked with:

print(retriever.action())

print(retriever.mood())

The above examples would print “The dog rolled over.” and “The golden-brown retriever is feeling excited.”.

When a method is called on an instance of a class in a hierarchy, Python goes into the class being called and searches for the method being called. If no method is found, Python goes into the next class in the hierarchy (the parent class) and searches for the method there. Python continues doing this until the desired method is found. In cases where multiple classes in a hierarchy contain methods with the same names, the first method found is the one used. Classes cannot access methods in classes below them in a hierarchy.

class Predator(Animal):

def \_\_init\_\_(self, size, color, mood, prey, hungry):

Animal.\_\_init\_\_(self, size, color, mood)

self.prey = prey

self.hungry = True

def eat(self):

return str(self) + “s eat”, prey + “s.”

def \_\_str\_\_(self):

return “< A”, str(self), “is a predator! >”

In the above example, the class Predator calls on the \_\_init\_\_ method from the parent class Animal to initiate an instance of itself. The \_\_init\_\_ method from the class Animal runs with the parameters passed to it (size, color, mood), and the additional attributes prey and hungry are created for the class Predator.

class Wolf(Predator):

def sound(self):

return “Howl!”

def action(self):

return “The wolf growled.”

def \_\_str\_\_(self):

return “< A”, str(self), “is a wolf! >”

In the above example, the class Wolf is a subclass of the class Predator, which is a subclass of the class Animal. Any instance of the class Wolf will have access to the methods in the classes Wolf, Predator, and Animal.

You can create subclasses that have more than one parent class by including the names of additional classes (separated by commas) in parentheses after the class name. In this case, Python will search for attributes or methods through the parent classes (and their respective hierarchies) from right to left:

class C(A, B):

def \_\_init\_\_(self):

self.etc = “etcetera”

In the above example, Python would search for attributes or methods by looking through the class A and all the classes above it in its hierarchy before moving onto B and all the classes above it in its hierarchy.

Besides just defining instance variables (or, attributes) in classes, you can create class variables that exist within classes but outside methods. These can be useful when you want to create variables that are part of a class that aren’t limited to the scope of any method within the class.

class Lemming(Animal):

number = 1

def \_\_init\_\_(self, size, color, mood)

Animal.\_\_init\_\_(self, size, color, mood)

self.ID = Lemming.number

number += 1

def sound(self):

return “Meep!”

def \_\_str\_\_(self):

return “< A”, str(self), “is a lemming! >”

In the above example, the class variable number is created by simply being defined outside a method (and conventionally, before \_\_init\_\_). Class variables can be accessed by typing the name of a class followed by a dot and the name of the class variable (in this case, Lemming.number). In the above example, whenever the \_\_init\_\_ function is run (when an instance of Lemming is created), the \_\_init\_\_ method from the class Animal is called, the attribute ID is bound to the current value of the class variable number, and number is incremented by 1. In this way, every time a new instance of Lemming is created, it has a unique value for the attribute ID.

If ever you need to confirm whether or not an object is an instance of a particular class, use the isinstance function:

isinstance(hippo, Animal)

The isinstance function accepts two arguments and returns True if the first argument (typically an object) is indeed an instance of the second argument (typically a class).

The pass keyword doesn’t tell Python to do anything (or, rather, it tells Python to do nothing!). It often serves as a placeholder for code in conditionals, loops, functions, or classes in Python:

class Reptiles(Animal):

pass

Python will treat the above example as a regular class that doesn’t do anything (without the pass keyword, Python may throw an error for not having anything in the body of the class).

**Generators**

A **generator** is a special type of function in Python that returns values in small parts rather than as a whole. They are written like regular functions with the exception of containing one or more yield statements in place of a return statement:

def print\_this():

n = 1

yield n

n += 1

yield n

n += 1

yield n

While a return statement returns a value and terminates a function, a yield statement returns a value and then pauses a function and saves the function’s state for later use. In the above example, every time a yield statement is reached, it will return a value and pause the function until the function is called on again, at which point the function will pick up where it left off and continue running until it meets the next yield statement.

Generators are run using the next() (or .\_\_next\_\_) method. The first time next() is called on a generator, the function is run until it reaches the first yield statement. The generator will pause and not continue running until it is called by another next() method, at which point it will pick up where it left off and repeat the same thing.

a = print\_this()

print(next(a)) # this will print, 1

print(next(a)) # this will print, 2

print(a.\_\_next\_\_()) # this will print, 3

If the generator is ever called more times than the yield statements it contains, it will raise a StopIteration error. Once a generator has been used to the end, it will need to be redefined for it to be reused (e.g. in the above case, a would have to be redefined as a = print\_this once again).

Generators can be implemented in loops:

def countdown(num):

print(“Initiating countdown sequence.”)

while num > 0:

yield num

num -= 1

print(“Blastoff!”)

The following is an example of the above generator being called:

blastoff = countdown(10)

blastoff # returns something of the sort, <generator object

countdown at 0x10213aee8>

next(blastoff) # prints, “Initiating countdown sequence.”

10

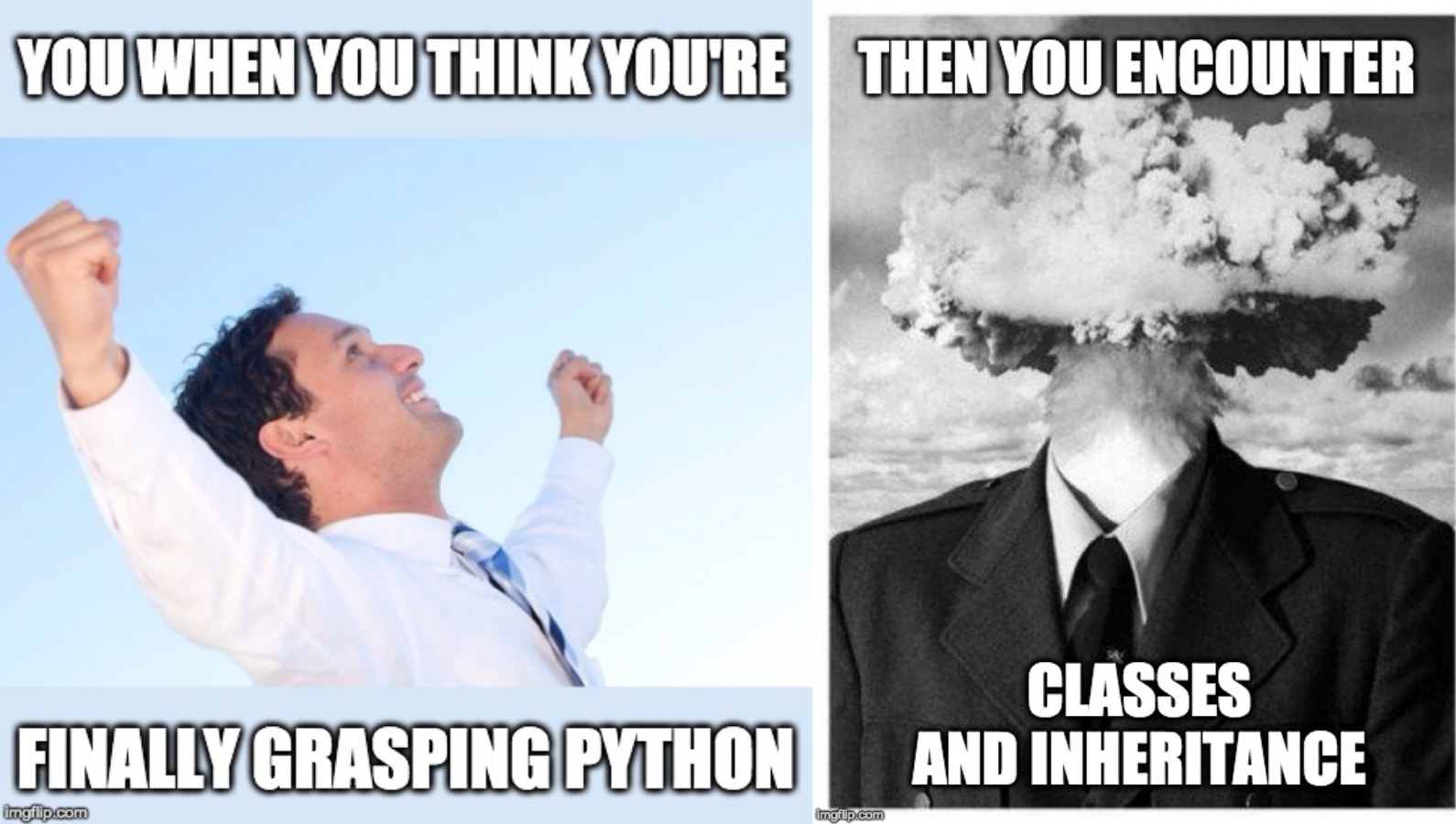
next(blastoff) # prints, 9

next(blastoff) # prints, 8

for loops can be used to run all of the yield statements in a generator at once. for loops typically won’t generate any StopIteration errors:

for iteration in countdown(10):

print(iteration)



## **WEEK 6**

**Algorithmic Complexity**

In programming, there can be multiple algorithms which can be used to accomplish the same task, albeit with varying levels of efficiency. An algorithm's efficiency is usually evaluated by its **order of growth**, or, the relationship between the size of the input it is given and the time it takes to complete the task. Several common types of orders of growth, including constant, linear, quadratic, logarithmic, and exponential orders of growth. Typically, the time a program takes to complete a task increases as the size of the given input increases.

**Big O notation** is a way to express the runtime of an algorithm as an order of growth. Typically, the worst-case scenario runtime of an algorithm is evaluated when using big O notation. When evaluating an algorithm’s order of growth this way, the factors that have the greatest impact on the algorithm’s runtime are often the factors that determine what order of growth the algorithm is categorized as. Smaller factors with lesser impact on an algorithm’s runtime (especially in relation to the larger factors) are often neglected.

To determine the Big O notation of an algorithm, construct an equation that represents the number of steps of an algorithm in relation to the size of the input (which is represented using *n*). What counts as a step in a program is somewhat variable, but often includes the following:

* Mathematical operations
* Comparisons
* Assignments
* Accessing of objects in memory

Then, simplify the equation based on the factors which have the most impact on the runtime of the algorithm. Additive and multiplicative constants (such as *5* or *5n*) are usually dropped, and the whole equation is simplified to a single term based on the dominant factors of the equation:

O(n2): n2 + 2n + 2

O(n2): n2 + 100000n + 31000

O(n): log(n) + n + 4

O(n log n): 0.0001\*n\*log(n) + 300n

O(3n): 2n30 + 3n

The above examples show the big O notation of various equations. The term representing the Big O notation is preceded by a capital *O* and surrounded by parentheses. The last example especially illustrates how much big O notation is influenced by the dominant term(s) in an algorithm (i.e. although 2n30 is a very large number, 3n has a greater impact on the algorithm’s runtime as n increases).

* In Big O notation, when two sequential terms are added to each other in an equation, the lower of the two terms is dropped and the higher kept (this is known as the *law of addition*). The *law of multiplication* (primarily used when dealing with nested for loops) states that when there is a for loop nested in another for loop, the number of steps in each of the for loops are multiplied by each other to simplify both terms into one term.
* Note: when counting “steps” in a program, a while loop is always counted as at least one step since it must be evaluated at least once to determine whether to run or not (even if it doesn’t actually satisfy the conditions to run). However, a for loop may not count as a step if there are no valid values for its counter variable to represent (as in for x in []: ... ).

The following are several common orders of growth (also known as classes of complexity) in order of complexity (from lowest to highest):

O(1) - constant runtime

O(log n) - logarithmic runtime

O(n) - linear runtime

O(n log n) - log-linear runtime

O(n2) - quadratic runtime

O(nc) - polynomial runtime (where c is a constant)

O(cn) - exponential runtime (where c is a constant)

**Simple Algorithms**

Some relatively simple algorithms include algorithms of the search and sort classes of algorithms. Search algorithms find a specific element in a larger collection of data, while sort algorithms organize collections of data. Search algorithms that are used on collections of sorted data are almost always faster than search algorithms that are used on unsorted collections of data.

The algorithm shown below is an example of a basic linear search algorithm that works on both sorted and unsorted collections of data:

def linear\_search(element, some\_list):

for item in some\_list:

if item == element:

return True

return False

The above example systematically checks each element of some\_list until a match with the parameter element is found. The algorithm’s worst-case runtime would occur if there were no matches of element in some\_list (*O: n*).

The algorithm shown below is a basic linear search algorithm that works on sorted collections of data:

def linear\_search\_2(element, some\_list):

for item in some\_list:

if item == element:

return True

elif item > element:

return False

return False

The above example systematically checks each element in some\_list until a match with the parameter element is found or an element in some\_list is found that is greater than the parameter element (at which point continuing the search would be pointless because all the other elements in the sorted collection of data would also be greater than the parameter element).

The algorithm shown below is an example of a bisection search algorithm that works on sorted collections of data:

def bisection\_search(element, some\_list):

high = len(some\_list)

low = 0

while True:

middle = (high + low) / 2

if some\_list[middle] == element:

return True

elif high == low or high == low + 1 or high == low - 1:

break

elif some\_list[middle] > element:

high = middle

elif some\_list[middle] < element:

low = middle

return False

The above example checks to see if the element at the midpoint of some\_list is the target element. If the element at the midpoint of some\_list is larger than the target element, the algorithm repeats the same process for the lower half of the list (since in a sorted list, the other half of the list will always be larger than the target element). If the opposite is true, the algorithm repeats the same process for the upper half of the list (since in a sorted list, the other half of the list will always be smaller than the target element). The algorithm repeats this process until the target element is found or the section of the list it is searching is of the length 1 (at which point the target element is not found and there is nothing more to search). Since with every iteration of the while loop the above algorithm divides the input it is searching in half, the worst-case runtime for the above algorithm would be the logarithm of the size of the input it is given (*O: log n*).

**Bubble sort** is the name of a simple sorting algorithm that sorts a collection of data. The algorithm works by iterating through the data comparing two adjacent elements at a time, and leaving them in place if they are sorted, or swapping their positions if they aren’t. This effectively “bubbles” the largest element in the data to the end of the data after one iteration, and the following iterations will continue to “bubble” the next largest elements in the data to the end of the data as well. This process is repeated until there are no more elements to sort in the data, and the data is then… sorted!

An example of a bubble sort algorithm is shown below:

def bubble\_sort(some\_list):

is\_sorted = False

while not is\_sorted:

is\_sorted = True

for index in range(1, len(some\_list):

if some\_list[index] < some\_list[index - 1]:

is\_sorted = False

temp = some\_list[index]

some\_list[index] = some\_list[index - 1]

some\_list[index - 1] = temp

In the worst-case scenario, all of the elements in some\_list would be in reverse order, and the algorithm would have to iterate through some\_list *n2* times to put all of the elements in their correct place (*O: n2*).

**Selection sort** is the name of another simple sorting algorithm that sorts a collection of data. The algorithm works by iterating through the data to find the smallest element and then swapping that element with the element at the beginning of the data. The algorithm then iterates through the data once again and finds the next-to-smallest element in the data and then swaps that element with the element in the place next to the beginning of the list. The algorithm repeats this process until the entire collection of data has been sorted.

The following is an example of a selection sort algorithm:

def selection\_sort(some\_list):

sorted\_index = 0

while sorted\_index != len(some\_list):

for index in range(sorted\_index, len(some\_list)):

if some\_list[index] < some\_list[sorted\_index]:

some\_list[sorted\_index], some\_list[index] = some\_list[index], some\_list[sorted\_index]

sorted\_index += 1

In the worst-case scenario, all of the elements in some\_list would be in reverse order, and the algorithm would have to iterate through some\_list *n2* times to put all of the elements in their correct place (*O: n2*).

**Merge sort** is the name of yet another simple sorting algorithm that sorts a collection of data. The algorithm works by splitting an unsorted collection of data into halves and further dividing those halves into halves and dividing those halves into halves, etc. until each element in the collection of data is a separate half. Then, the algorithm merges all of those halves into pairs (or rather, sorted collections) and merges those pairs into pairs, etc. until the entire collection of data is sorted.

Merge sort merges collections of data by comparing the first element in one half of a sorted collection of data to the first element of an adjacent half of a sorted collection data and placing the smaller of the elements in a new, merged, sorted collection of data. Then the algorithm compares the next element in one half of a sorted collection of data to the element of the adjacent half of a sorted collection data and placing the smaller of the elements in the new, merged, sorted collection of data. Merge sort repeats this process until all the elements in both halves of the collections of data are stored and sorted the new, merged collection of data.

## **WEEK 7**

**Plotting Graphs**

There are a few modules included with Python that are able to construct graphs and plot points of data on them. One of these modules is named pylab.

To use pylab, simply import it using the import statement:

import pylab

To construct and plot points of data along a graph using pylab, use the .plot() function and provide the associated x and y values:

x\_values = []

y\_values = []

for i in range(30):

x\_values.append(i)

y\_values.append(i\*\*2)

pylab.plot(x\_values, y\_values)

The above example plots a simple square function.

If you continue to construct additional graphs, their data will be plotted on the same window as your first graph. To create separate windows for separate graphs, use the .figure() function and provide the name you would like the graph to be referenced by, followed by a .plot() function of the graph:

y\_values\_2 = []

for i in range(30):

y\_values\_2.append(i\*\*3)

pylab.figure(“another graph”)

pylab.plot(x\_values, y\_values\_2)

The above example constructs a separate graph (of a simple cubed function) that can be referenced again using the name another graph.

You can add titles and x-axis and y-axis labels to your graphs using the .xlabel() and .ylabel() and .title() functions:

y\_values\_3 = []

for i in range(30):

y\_values\_3.append(i+5)

pylab.figure(“yet another graph”)

pylab.plot(x\_values, y\_values\_3)

pylab.xlabel(“these are the x values”)

pylab.ylabel(“these are the y values”)

pylab.title(“A Simple Linear Graph”)

The above example constructs a graph (a simple linear function) and gives a title and some labels to a graph.

To manipulate graphs that have already been constructed and named, use the .figure() function again. All functions or methods in pylab that relate to graphs apply to the first graph created, or if the .figure() function has been used, to the graph specified by the most recent .figure() function. To clear a graph of its points and all other data (i.e. to make a new slate), use the .clf() function:

pylab.figure(“another graph”)

pylab.clf()

The above example clears the graph named another graph. Functions from pylab that are called apply to the graph referenced by the last .figure() statement.

You can add labels to the points plotted on a graph (using the label parameter of the .plot() function) and instruct pylab to display a legend showing those labels (using the .legend() function). You can also limit the range of the axes of the graph using the .xlim() and .ylim() functions:

pylab.figure(“yet another other graph”)

pylab.ylim(0, 1000)

pylab.plot(x\_values, y\_values, label=“square function”)

pylab.plot(x\_values, y\_values\_2, label=“cubed function”)

pylab.legend(loc=“upper left”)

pylab.title(“A Simple Linear Graph”)

The above example limits the y-axis of a graph to 1000, plots a square function and a cubic function on the graph, assigns separate labels to each of them, and instructs pylab to display a legend in the upper left corner of the window.

There are several ways to customize the way a graph is displayed. You can change the color and method of displaying the data in a graph by adding a string as the third argument to the function .plot(). The first character of the argument specifies what color the data should be. The next characters specify how the data should be displayed:

pylab.figure(“a colorful graph”)

pylab.plot(x\_values, y\_values, “bo”, label=“square function”, linewidth=2.0)

pylab.plot(x\_values, y\_values\_2, “g^”, label=“cubed function”, linewidth=3.0)

pylab.legend(loc=“upper left”)

The above example constructs a graph of a square function displayed by blue circles (b for blue and o for circles) and a cubed function displayed by green triangles (g for green and ^ for triangles). Additionally, the .plot() function can take another argument called linewidth that specifies how thick the line (or shapes that make up the line) should be displayed (in pixels). See the pylab documentation for more coloring and display options.

Finally, you can specify how you want separate graphs to be plotted (within the same window) by using the .subplot() function. The .subplot() function accepts an integer argument of which the first digit determines how many rows of graphs should be displayed, the second determines how many columns of graphs should be displayed, and the last specifies the order of the graph of the next .plot() function:

pylab.figure(“two graphs in one”)

pylab.subplot(211)

pylab.ylim(1000)

pylab.plot(x\_values, y\_values)

pylab.subplot(212)

pylab.ylim(1000)

pylab.plot(x\_values, y\_values2)

pylab.yscale(“log”)

The above example creates a window with two graphs (one of a square function, the other of a cubed function, both with a y-axis limit of 1000) and constructs each graph on a separate row in a single column (2 for the number of rows, 1 for the number of columns, and 1 and 2 for each of the graphs, respectively). The additional function call (.yscale()) specifies to display the y-axis as a logarithmic scale (“log”) instead of a linear scale (which can be useful in some instances).

Enjoy 6.00.1x!

ApplePieGiraffe