Automate the Boring Stuff

Who is this course for?

On its own, this book won’t turn you into a professional software developer any more than a few guitar lessons will turn you into a rock star. But if you’re an office worker, administrator, academic, or anyone else who uses a computer for work or fun, you will learn the basics of programming so that you can automate simple tasks such as these:

* Moving and renaming thousands of files and sorting them into folders
* Filling out online forms—no typing required
* Downloading files or copying text from a website whenever it updates
* Having your computer text you custom notifications
* Updating or formatting Excel spreadsheets
* Checking your email and sending out prewritten responses

These tasks are simple but time-consuming for humans, and they are often so trivial or specific that there’s no ready-made software to perform them. Armed with a little bit of programming knowledge, however, you can have your computer do these tasks for you. Example:

➊ passwordFile = open('SecretPasswordFile.txt')  
➋ secretPassword = passwordFile.read()  
➌ print('Enter your password.')  
   typedPassword = input()  
➍ if typedPassword == secretPassword:  
   ➎ print('Access granted')  
   ➏ if typedPassword == '12345':  
       ➐ print('That password is one that an idiot puts on their luggage.')  
  else:  
   ➑ print('Access denied')

You might not know anything about programming, but you could probably make a reasonable guess at what the previous code does just by reading it. First, the file SecretPasswordFile.txt is opened ➊, and the secret password in it is read ➋. Then, the user is prompted to input a password (from the keyboard) ➌. These two passwords are compared ➍, and if they’re the same, the program prints Access granted to the screen ➎. Next, the program checks to see whether the password is 12345 ➏ and hints that this choice might not be the best for a password ➐. If the passwords are not the same, the program prints Access denied to the screen ➑.

What is Python?

*Python* is a programming language (with syntax rules for writing what is considered valid Python code) and the Python interpreter software that reads source code (written in the Python language) and performs its instructions.

# Cheat Sheet

|  |  |
| --- | --- |
| **Command** | **Description** |
| Ctrl + [ / ] | Indentation on left / right respectively |
| // | Floor division e.x. 3 // 2 = 1 |
|  |  |
|  |  |
|  |  |

# Chapter 1 – Python Basics

REPL – Interactive shell which stands for Read Evaluate Print loop.

Graphical user interface, text, application

Description automatically generated

Figure 1 - First Python expression

2 + 2. 2 is a value while + is an operator. Expression will contain values and operators, which will result in an evaluation. Just typing 4 on it’s own and pressing Enter will also result in an evaluation, where 4 simply evaluates itself to 4.

Graphical user interface, application

Description automatically generated

Figure 2 - Math Operators from highest to lowest precedence

Precedence refers to the order of operations similar to mathematics.

## String Operations

**String example**: >>> ‘Hello, world!’

## String Concatenation and Replication

‘Alice’ + ‘Bob’ = AliceBob

In Python, it is possible to perform a **string replication** operation as follows:

‘Alice’ \* 5 will result in Alice printed five times.

## Variables

>>> spam = 40

>>> spam

40

A variable is *initialised* the first time a value is stored in it.

Graphical user interface, application

Description automatically generated

Figure 3 - Valid and Invalid Variable Names

Variables are case sensitive.

**Taking input and printing integers**

print('What is your age?')    # ask for their age  
   myAge = input()  
   print('You will be ' + str(int(myAge) + 1) + ' in a year.')

***Note***: int type cast variable to an integer for + 1 to be applied to it otherwise Python will think you are trying to concatenate with a string, as that is the only operation possible. Then the whole expression is converted into a string for concatenation to occur. Python gives an error because the + operator can only be used to add two integers together or concatenate two strings. You can’t add an integer to a string, because this is ungrammatical in Python. You can fix this by using a string version of the integer instead, as explained in the next section.

**Comment**

# This line is a comment

# Chapter 2 – Flow Control

Flow control statements can decide which Python instructions to execute under which conditions.

In Python, for **Boolean** variables, this is how you define their state, always start with capital T or F:

* spam = True
* apple = False

**Comparison operators** are exactly as the other languages. Even escape character \

Ex.

print('42 == \'42\' evaluates to ' + str(42 == '42'))

## Binary Boolean Operators

In Python, and or not Boolean operators are written as literals.

Ex. True and True

True

(4 < 5) and (9 < 6)

False

(1 == 2) or (2 == 2)

True

not True

False

## Blocks of Code

Grouping lines of code together is possible through indentation. There are three rules for blocks:

1. blocks begin when the indentation increases
2. blocks can contain other blocks
3. blocks end when the indentation decreases to zero or to a containing block’s indentation

## if statements

In Python the if statement is a bit different, rawer then other languages. Indentation plays an important part here. Syntax wise it is quite relaxed. *Also notice no semicolon anywhere.*

Ex.

If name == ‘Alice’:

Print(‘Hi Alice!’)

Elif age < 12:

Print(‘something else’)

Else:

Print(‘Unknown’)

## while statements

spam = 0  
while spam < 5:  
    print('Hello, world.')  
    spam = spam + 1 # spam+=1

continue statement

  while True:  
      print('Who are you?')  
      name = input()  
      if name != 'Joe':  
         continue  
       print('Hello, Joe. What is the password? (It is a fish.)')  
      password = input()  
       if password == 'swordfish':  
          break  
 print('Access granted.')

for loop and range function

print('My name is')  
for i in range(5):  
    print('Jimmy Five Times (' + str(i) + ')')

i is set to 0 and the value in range will go up to but not including this value. So 0-4

for I in range(12,16)

print(i)

result: 12,13,14,15

for I in range(0, 10, 2) will count zero to eight ( since 10 not inclusive) in intervals of 2

importing modules

import random, clock

**ending a program early - sys.exit()**

# Chapter 3 – Functions

## Defining a function

Def hello():

Print(‘Hello there’)

hello() # this will be a function call

helloUser(name)

# passing a value to the variable name, which make it an argument of the function helloUser()

**return keyword**

The return keyword allows the programmer to return a value from or the expression evaluation within a function. In Python, there is no need to declare what is expected as a return type for the defined function as can be seen below:

import random

def getAnswer(answerNumber):

       if answerNumber == 1:

           return 'It is certain'

       elif answerNumber == 2:

           return 'It is decidedly so'

       elif answerNumber == 3:

           return 'Yes'

       elif answerNumber == 4:

           return 'Reply hazy try again'

       elif answerNumber == 5:

           return 'Ask again later'

       elif answerNumber == 6:

           return 'Concentrate and ask again'

       elif answerNumber == 7:

           return 'My reply is no'

       elif answerNumber == 8:

           return 'Outlook not so good'

       elif answerNumber == 9:

           return 'Very doubtful'

r = random.randint(1, 9)

fortune = getAnswer(r)

print(fortune)

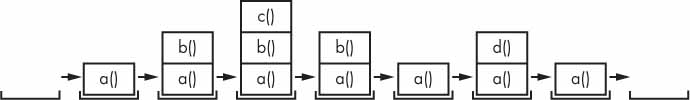
**None value**

None represents the absence of a value. The none value is the only value of the *Nonetype* data type. This is quite useful when you need to store something that won’t be confused for a real value in a variable.

**Note:** behind the scenes Python add a return None to the end of any function definition with no *return* statement. If the return statement is used without a value, then *None* is returned.

## The Call Stack

Similar to a long conversation with several details involving many subjects, calling a function does not send the execution on a one-way trip to the top of a function. Python will remember which line of code called the function so that the execution can return there when it encounters a *return* statement. If that original function called other functions, the execution would return to those functions first, before returning from the original function call.



### Local and Global Scope

Variable in global scope cannot use variables within a local scope. On the other hand, local scope functions can use global scope variables. You can use the same name for a local variable and a global variable. This can be confusing, and that is why the **global** statement exists. The global statement allows a user to specify that the global declared variable will be used.

ex.

def spam():  
   global eggs  
   eggs = 'spam'  
  
eggs = 'global'  
spam()  
print(eggs)

Output: spam

**Note**

If you try to use a local variable in a function before you assign a value to it, as in the following program, Python will give you an error.

def spam():

    print(eggs)

    eggs = 'spam local'

eggs = 'global'

spam()

This error happens because Python sees that there is an assignment statement for eggs in the spam() function and, therefore, considers eggs to be local. But because print(eggs) is executed before eggs is assigned anything, the local variable eggs doesn’t exist. Python will not fall back to using the global eggs variable.

## Exception Handling

In real life you want programs to detect and handle errors. We can use the **try** and **except** statement to handle these scenarios. For instance, the divide by zero case can be handled as follows:

def spam(divideBy):

try:

return 42/divideBy

except ZeroDivisionError:

print(‘Error: Invalid Argument.’)

print(spam(2))

print(spam(12))

print(spam(0))

print(spam(1))

This code will catch the error and return both the error text and *None* as that is the default return type. The try and except method can also catch errors in the try statement. If we place the print statements inside the try and define spam as just a division operation, we get a cleaner output and program stops executing at the first caught error.

# Chapter 4 – Lists

## List data type

The list is a value that contains multiple values in an ordered sequence. Values inside a list are called items.

ex. [‘dog’,’cat’]

**Getting values from a list**

Like accessing an array.

ex. spam = [‘cat’,’sheep’]

spam[0] # output cat

spam[1] # output sheep

Note: if you try to print a value which is outside the bounds of the list, Python will err out with an *IndexError* message.

**Listception**

It is possible to have a list that contains lists, which is known as a double array, double list or matrix from other languages.

ex. spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]

spam[0][1] # output bat

spam[1][4] # output 50

**Negative Index**

Contrary to other languages, it is possible to use a negative index which will start counting from the end of the list. For instance, -1 will be the last index in a list, while -2 will be second to last.

**Getting a list from another list with slices**

Just as an index can get a single value from a list, a *slice* can get several values from a list, in the form of a new list.

ex. spam[1:4]

Note: including first value, up to and excluding second value. The slice forms a new list.

**More list operations in Chapter3.py**

**Multiple Assignment Trick (Tuple Unpacking)**

Python has a neat way of handling unpacking information from a list. Instead of assigning each value to a variable one after the other, you can use the shortcut as follows. Note that the number of variables and the length of the list must be exactly equal.

ex. cat = [‘fat’, ‘gray’, loud]

size = cat[0]

color = cat[1]

disposition = cat[2]

*instead we can write it as*

size, color, disposition = cat

## Methods

A *method* is the same thing as a function except it is called on a value. For instance, spam.index(‘hello’) is a method that takes hello as input and gives us the index if it lies within the spam list.

Note**:** if a value is not within a list, a *ValueError* error is returned. When there are duplicates of the value in the list, the index of the first appearance is returned.

## Mutable and Immutable Data Types

While a list and a string are similar in the way they are stored and can be accessed when it comes to indexes, they have one important difference. A list value is a *mutable* data type, it can have values added, removed and changed. A string is *immutable*, which means it cannot be changed. For instance, you cannot simply get a character and reassign it to a new character. The only way to modify a string is to create and copy the old to a new string using slicing and concatenation.

## Tuple Data Type

The main difference between a list and a tuple, apart from the parentheses which are round brackets “()” for the former, is that a tuple is **immutable**. This means that its value much like a string cannot be changed, appended or modified.

Ex. eggs = (‘hello’, 42, 0.5)

Bread = (‘hello’,) # perfectly acceptable indicates a tuple with a single value

**Converting types with list and tuple**

It is possible to convert between list and tuple and this comes in handy if you need a mutable version of a tuple value.

## References

When you assign 42 to a *spam* variable, you are actually creating the 42 value in memory and storing the reference to it in the spam variable. If you then create *cheese* variable, and assign it to *spam*, you are actually copying the reference to it. Both spam and cheese now have a reference to 42. If you then decide to assign 100 to spam, spam now stores a new reference to the new value of 100. This means now spam is referring to 100 but cheese still refers to 42 and is not affected. Integers are **immutable** and don’t change. On the other hand, lists are mutable.

## Identity and id() function

All values in Python have a unique identity that can be obtained with the id() function. When you are adding to a string, you are basically creating a new string object in a different place in memory, and the variable refers to the new string. In the case of lists, given they are mutable objects, appending changes the existing list object. This is known as ***modifying in place***. Common list methods such as append, extend, remove, sort and reverse among others modify their lists in place.

Python’s automatic garbage collector deletes any values not being referred to by any variables to free up memory.

## Copy and deepcopy functions

We can use copy.copy(*listname*) to copy the values of a list instead of just assigning a new variable to an existing list, and therefore simply passing a reference or the memory address of where it is at. Essentially in this way we would have two unique lists with the same values.

NOTE: **% width can be used for matrix wraparound when required in operations**

# Chapter 5 – Dictionaries and Structuring Data

## Dictionary Data Type

Like a list, a dictionary is a mutable collection of many values. The major difference is that indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called *keys*, and a key with an associated value is called a *key-value pair*.

*myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}*

The *myCat* variable has now been assigned a dictionary, where the key are size, color and disposition.

*myCat[‘size’]*

will result in fat

Integers can also be a valid key for dictionaries; however they do not need to start at 0.

*spam = {12345: 'Luggage Combination', 42: 'The Answer'}*

**Dictionaries vs Lists**

Unlike lists, items in a dictionary are unordered. There is no “first” item in a dictionary. While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary.

Text

Description automatically generated

### keys(), values() and items() Methods

There are three dictionary methods that will return list-like values of the dictionary’s keys, values, or both keys and values: *keys(), values() and items().* Notice that the values in the dict\_items value returned by the items() method are tuples of the key and value. If you want a true list from one of these methods, pass its list-like return value to the list() function.

### get() method

The get method for dictionaries allows us to check if a value exists and output something in the case it does not exist in the dictionary. This saves us time and makes the code more readable.

picnicItems = {'apples': 5, 'cups': 2}

'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'

### setdefault()

You will often have to set a value in a dictionary for a certain key only if that key does not already have a value. The setdefault() method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist. If the key does exist, the setdefault() method returns the key’s value.

**Pretty Printing using pprint**

It is possible to print dictionaries and nested key value pairs in a clear format using the library ***pprint***. Remember to import the library first before using it. For printing, the method pprint can be used taking a dictionary as an argument.

pprint.pprint(picnicItems)

If you want to obtain the prettified text as a string value, you can call the format method.

pprint.pformat(picnicItems)

## Nested Dictionaries and Lists

Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values.

allGuests = {'Alice': {'apples': 5, 'pretzels': 12},  
             'Bob': {'ham sandwiches': 3, 'apples': 2},  
             'Carol': {'cups': 3, 'apple pies': 1}}  
  
def totalBrought(guests, item):  
    numBrought = 0  
  ➊ for k, v in guests.items():  
      ➋ numBrought = numBrought + v.get(item, 0)  
     return numBrought  
  
print('Number of things being brought:')  
print(' - Apples         ' + str(totalBrought(allGuests, 'apples')))  
print(' - Cups           ' + str(totalBrought(allGuests, 'cups')))  
print(' - Cakes          ' + str(totalBrought(allGuests, 'cakes')))  
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches')))  
print(' - Apple Pies     ' + str(totalBrought(allGuests, 'apple pies')))

Inside the totalBrought() function, the for loop iterates over the key value pairs in guests. Inside the loop, the string of the guest’s name is assigned to *k* and the dictionary of the picnic items they are bringing is assigned to *v*. If the item parameter exists as a key in this dictionary, its value (the quantity) is added to numBrought. If it does not exist as a key, the get() method returns 0 to be added to numBrought.