Python

INTRODUCTION

Python is a simple, easy to learn programming language that bears some resemblance to shell script. It has gained popularity very quickly due to its shallow learning curve. It is supported on all operating systems. https://www.python.org/

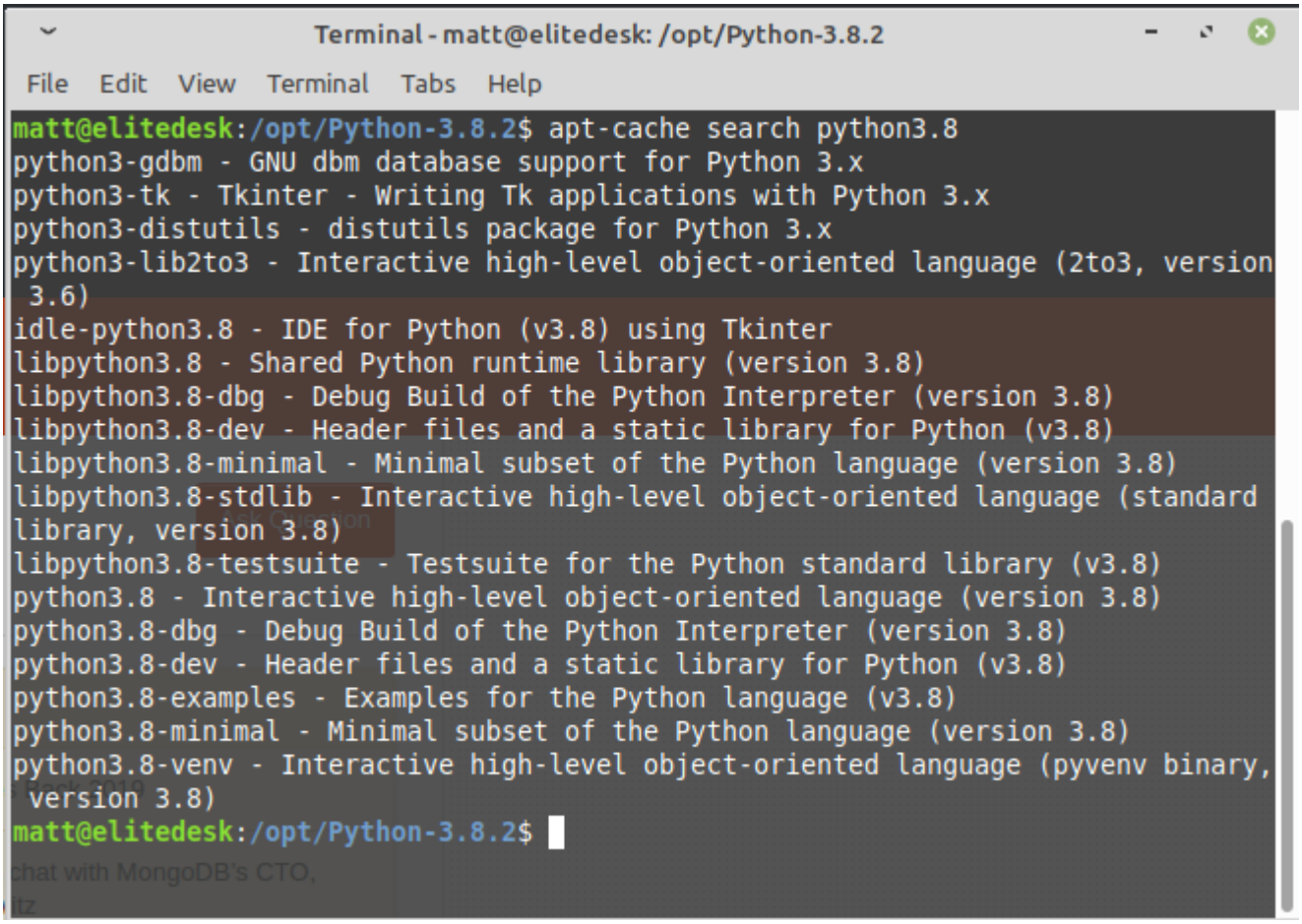
INSTALLATION

Installers for all operating systems are available here and on linux it tends to be installed by default in most distributions. This is quickly and easily checked by using the **python3 -V** command.

```
matt@elitedesk:~/Documents/Source Code/Python-3.8.2$ python3 -V
Python 3.6.9
```

You may find that the version installed in your distribution

lags slightly behind the very latest available from python.org. If you want to install the very latest version, then you can either download the source code and compile it, or add the repository and install it using your package management system. Check the version you want isn't already included in your package management system first using **apt-cache search python3.8**



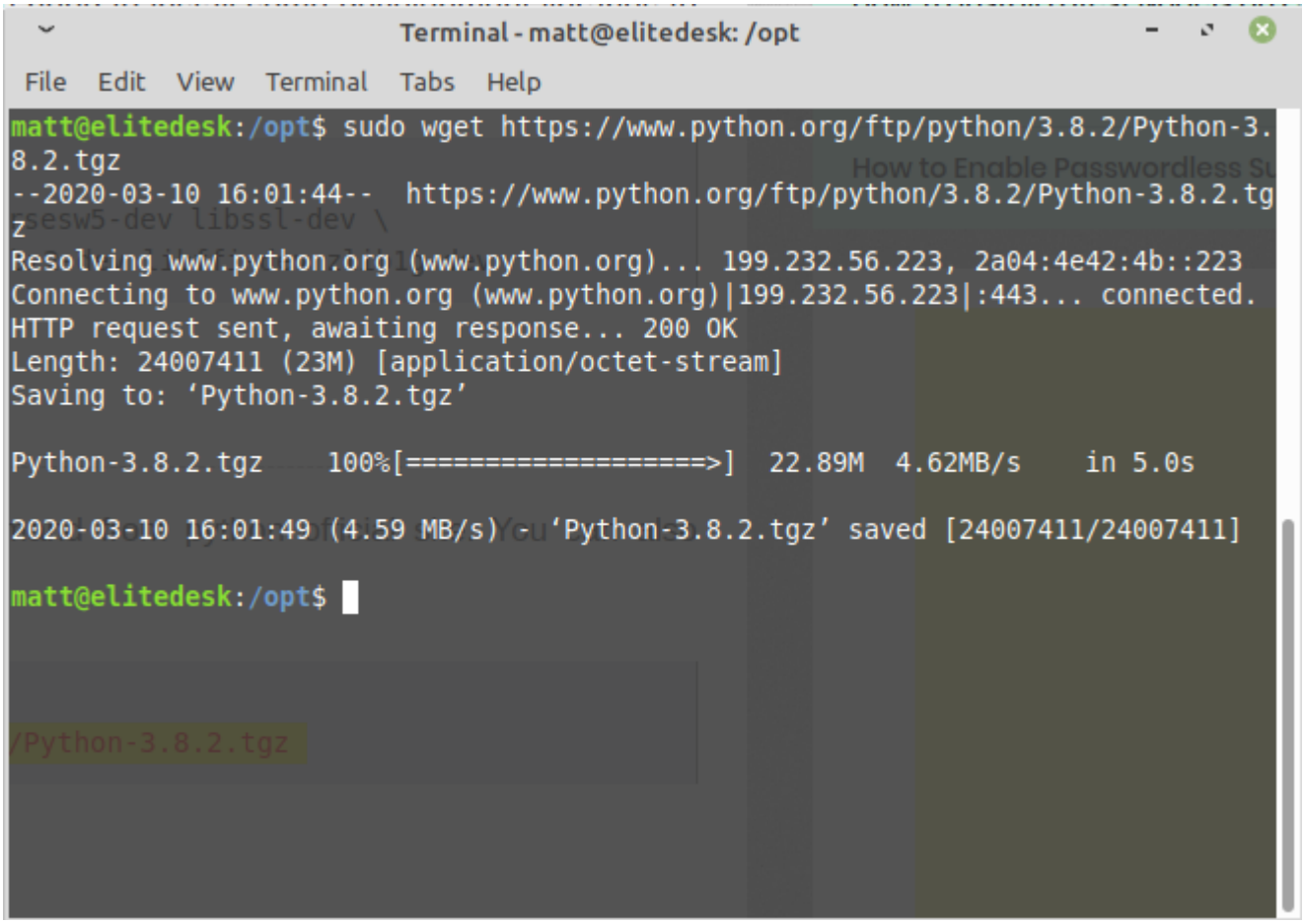
INSTALLATION VIA SOURCE CODE *(debian based distributions)*

Ensure the pre-requisites are installed first from the distro's default repo's.

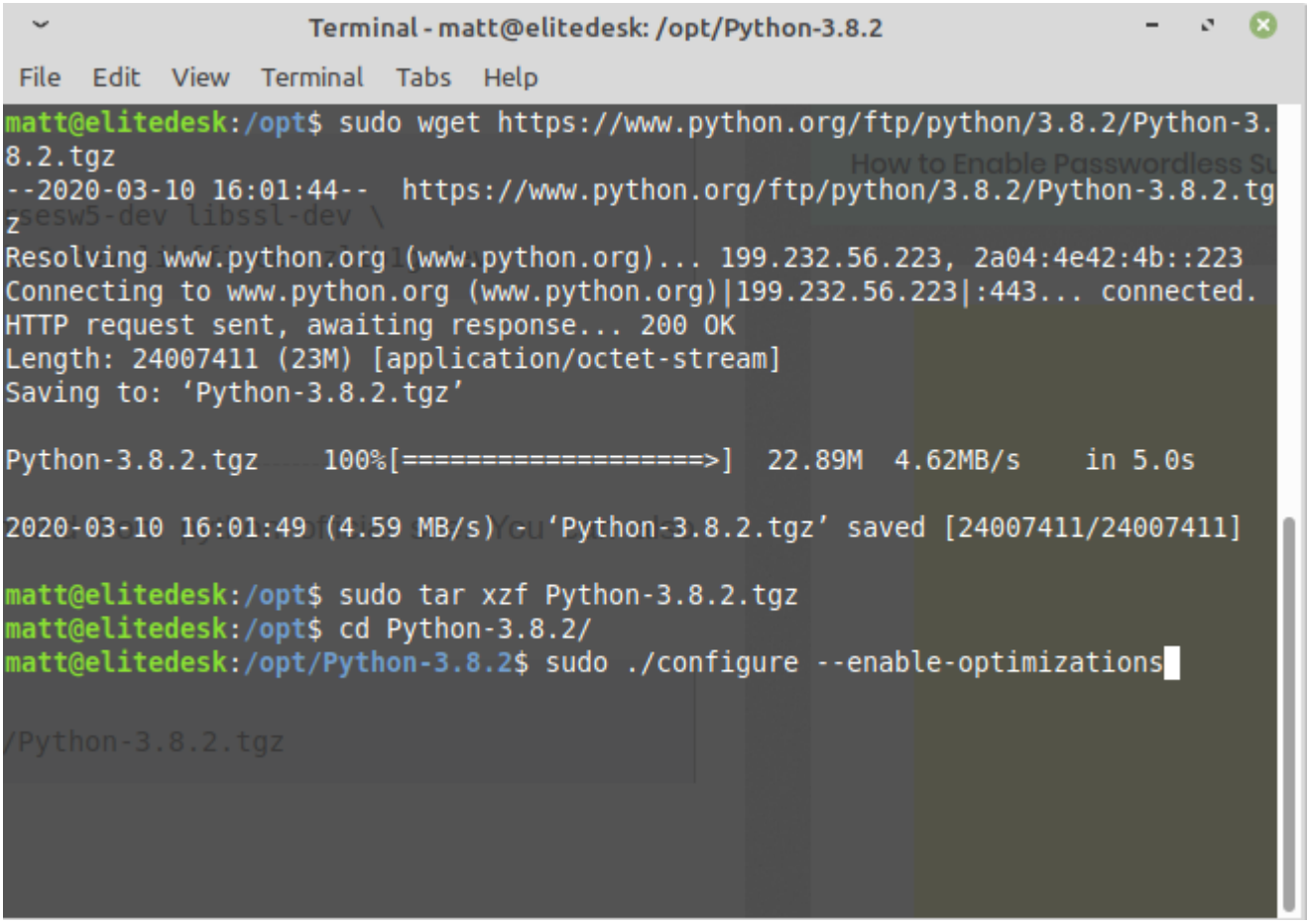
sudo apt-get install build-essential checkinstall

sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev

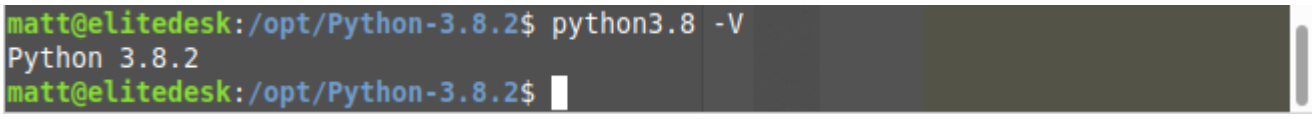
Download the source code from here or use **wget** and unzip using ***tar -zxvf Python-3.8.2.tgz***



Download the python source code from the command line using **sudo wget https://www.python.org/ftp/python/3.8.2/Python-3.8.2.tgz**



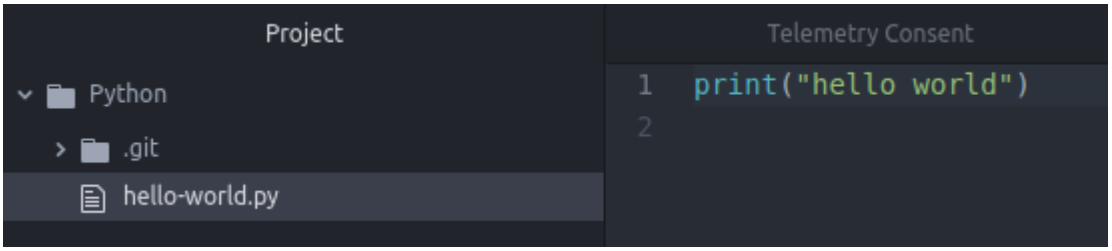
Extract the archive using **tar xzf Python-3.8.2.tgz**
cd into **Python-3.8.2** directory
sudo ./configure –enable-optimizations to create makefile
sudo make altinstall to install python without overwriting the version already installed in **/usr/bin/python** by your distro



Check python version with the **python3.8 -V** command

EXECUTING A PYTHON SCRIPT

Before getting into coding in python, I'll put this section in here just to satisfy your curiosity about how you actually execute a python script, since python ain't shell script…



The simplest of all python scripts? The simple “hello world” script **hello-world.py**

```
matt@elitedesk:~/Documents/Scripts/Python$ ./hello-world.py
bash: ./hello-world.py: Permission denied
matt@elitedesk:~/Documents/Scripts/Python$ chmod +x hello-world.py
matt@elitedesk:~/Documents/Scripts/Python$ ./hello-world.py
./hello-world.py: line 1: syntax error near unexpected token `"hello world"'
./hello-world.py: line 1: `print("hello world")'
matt@elitedesk:~/Documents/Scripts/Python$
```

Attempting to execute a python script like you would a shell script doesn’t end well. Python ain’t Shell after all.

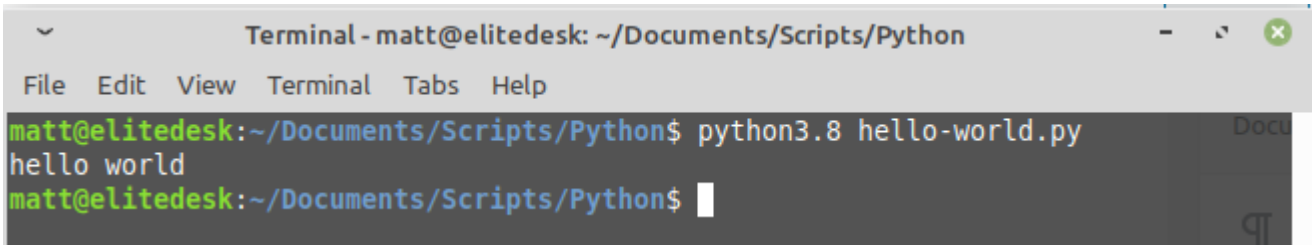
The hint was in the use of the python3.8 -V command previously in order to check the version of python i.e. to execute your python script using python 3.8.2, you could use the command **python3.8 hello-world.py**

PYTHON PROGRAMMING

COMMENTS

Comment code or your own comments throughout your python code for readability by placing a # at the front of the line. The python interpreter will ignore any lines beginning with a hash symbol. Alternatively, use a triple quote, e.g. ”’ but hashes are the official method of commenting a line of code/notes.

WORKING WITH STRINGS



hello world.

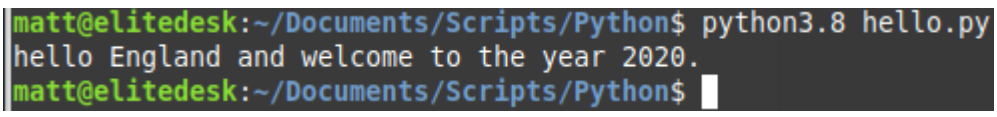
print("hello world") -prints the output *hello world* to the screen

country_name = "England" -create a variable country_name and assign string value of England

number_of_years = 2020 -create a variable number_of_years and assign numeric value of 2020

brexit_event = True -create a boolean variable with a true or false value

print("hello " + country_name + " and welcome to the year " + str(number_of_years) -Note that you can't concatenate a string and an integer so you need to convert the integer to a string using the **str()** function



executing the hello.py script comprised of the three lines above

print ("Cyberfella Limited"\n"python tutorial") -puts a new line between the two strings

print (“Cyberfella\\Limited”) -escape character permits the print of the backslash or other special character that would otherwise cause a syntax error such as a “

phrase = “CYBERFELLA”

print (phrase.lower()) -prints the string *phrase* in lowercase. There are other functions builtin besides upper and lower to change the case.

print (phrase.islower()) -returns False if the string phrase is not lower case

print (phrase.lower().islower()) -returns *True* after converting the uppercase string *phrase* to lowercase.

print (len(phrase)) -returns the length of the string, i.e. 10

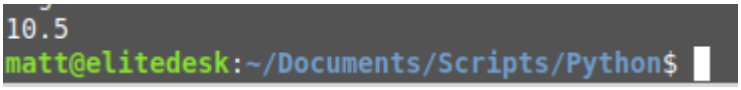
print (phrase[0]) -returns the first letter of the string, i.e. C

print (phrase.index(“Y”)) -returns the location of the first matching parameter in the string i.e. 1 Note you can have a string as a parameter e.g. CYB

print (phrase.replace(“FELLA”,”FELLA LTD”)) -replaces matching part of the string (substring) FELLA with FELLA LTD

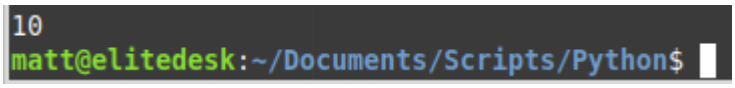
WORKING WITH NUMBERS

Python can do basic arithmetic as follows, **print (2.5 + 3.2 * 5 / 2)**



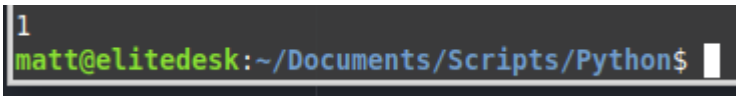
python performing arithmetic 2.5 + 3.2 * 5 / 2 = 10.5 based on the PEMDAS order of operations. The "**operations**" are addition, subtraction, multiplication, division, exponentiation, and grouping; the "**order**" of these **operations** states which **operations** take precedence (are taken care of) before which other **operations**. … Multiplication and Division (from left to right) Addition and Subtraction (from left to right)

To change the order of operations, place the higher priority arithmetic in parenthesis, e.g. **print (2 * (3 + 2))**



3+2 = 5 and 2 * 5 = 10. The addition is prioritised over the multiplication by placing the addition in () to be evaluated first.

print (10 % 3) is called the Modulus function, i.e. 10 mod 3. This will divide 10 by 3 and give the remainder, i.e. 10 / 3 = 3 remainder 1. So it outputs 1.



How to perform a MOD function, i.e. give the remainder when two numbers are divided. e.g. 10 % 3 (10 mod 3) = 1

The absolute value can be obtained using the abs function, e.g. **print (abs(my_num))**

```
98
matt@elitedesk:~/Documents/Scripts/Python$
```

The variable my_num had been assigned a value of -98

print (pow(3,2)) prints the outcome of 3 to the power of 2, i.e. 3 squared.

```
9
matt@elitedesk:~/Documents/Scripts/Python$
```

3 squared is 9

print (max(4,6)) prints the larger of the two numbers, i.e. 6 **min** does the opposite

```
6
matt@elitedesk:~/Documents/Scripts/Python$
```

6 is larger than 4

print (round(3.7)) rounds the number, i.e. 4

```
4
matt@elitedesk:~/Documents/Scripts/Python$
```

3.7 rounded is 4

There are many more math functions but they need to be imported from the external *math* module. This is done by including the line **from math import** * in your code

print (floor(3.7)) takes the 3 and chops off the 7 (rounds

down) **ceil** does the opposite (rounds up)

```
3
matt@elitedesk:~/Documents/Scripts/Python$
```

The **floor** function imported from the **math** module, returns the whole number but not the fraction

print (sqrt(9)) returns the square root of a number, i.e. 3.

```
3.0
matt@elitedesk:~/Documents/Scripts/Python$
```

the square root of 9 is 3.0 according to pythons sqrt function

GETTING INPUT FROM USERS

name = input ("Enter your name: ") will create a variable *name* and assign it the value entered by the user when prompted by the *input* command.

num1 = input ("Enter a number: ")

num2 = input ("Enter another number: ")

Any input from a user is treated as a string, so in order to perform arithmetic functions on it, you need to use int or float to tell python to convert the string to an integer or floating point number if it contains a decimal point.

result = int(num1) + int(num2) or **result = float(num1) + float(num2)**

print (result)

WORKING WITH LISTS

Multiple values are stored in square brackets, in quotes separated by commas,

friends = [“Eldred”, “Chris”, “Jules”, “Chris”]

You can store strings, numbers and booleans together in a list,

friend = [“Eldred”, 1, True]

Elements in the list start with an index of zero. So Chris would be index 1.

print(friends[1])

Note that **print (friends[-1])** would return the value on the other end of the list and **print (friends[1:])** will print the value at index 1 and everything after it. print (friends[1:3]) will return element at index 1 and everything after it, up to but not including element at index position 3.

To replace the values in a list,

friends[1] = “Benny” would replace Chris with Benny

USING LIST FUNCTIONS

lucky_numbers = [23, 8, 90, 44, 42, 7, 3]

To extend the list of *friends* with the values stored in *lucky_numbers*, effectively joining the two lists together,

friends.extend(lucky_numbers) Note that since Python3 you’d need to use **friends.extend(str(lucky_numbers))** to convert the integers to strings before using functions such as sort or you’ll receive an error when attempting to sort a list that is a mix of integers and strings.

To simply add a value to the existing list

friends.append(“Helen”)

To insert a value into a specific index position,

friends.insert(1, “Sandeep”)

To remove a value from the list,

friends.remove(“Benny”)

To clear a list, use **friends.clear()**

To remove the last element of the list, use **friends.pop()**

To see if a value exists in the list and to return its index value,

print (**friends.index(“Julian”))**

To count the number of similar elements in the list,

print (friends.count(“Chris”))

To sort a list into alphabetical order,

friends.sort()

print (friends)

To sort a list of numbers in numerical order,

lucky_numbers.sort()

print (lucky_numbers)

To reverse a list, **lucky_numbers.reverse()**

Create a copy of a list with,

friends2 = friends.copy()

WORKING WITH TUPLES *(pronounced tupples)*

A tuple is a type of data structure that stores multiple values but has a few key differences to a list. A tuple is immutable. You can’t change it, erase elements, add elements or any of the above examples of ways you can manipulate a list. Once set, that’s it.

Create a tuple the same way you would a list, only using parenthesis instead of square brackets,

coordinates = (3, 4)

print (coordinates[0]) returns the first element in the tuple, just as it does in a list.

Generally, if python stores data for any reason whereby it doesn’t stand to get manipulated in any way, it’s stored in a tuple, not a list.

FUNCTIONS

Just as with Shell Scripting, a function is a collection of code that can be called from within the script to execute a sequence of commands. function names should be in all lowercase and underscores are optional if you want to see a space in the function name for better readability, e.g. hello_world or helloworld are both acceptable.

def hello_world ():

print (“Hello World!”)

commands inside the function MUST be indented. To call the function from within the program, just use the name of the function followed by parenthesis, e.g.

hello_world()

You can pass in parameters to a function as follows,

def hello_user (name):

print (“Hello ” + name)

pass the name in from the program with **hello_user(“Bob”)**

def hello_age (name, age):

print (“Hello ” + name = ” you are ” + str(age))

hello_age (“Matt”, 45)

RETURN STATEMENT

In the following example, we’ll create a function to cube a number and return a value

def cube (num);

return (num ^3)

Call it with **print (cube(3))**. Note that without the return statement, the function would return nothing despite performing the math as instructed.

result = cube(4)

print (result)

Note that in a function that has a return statement, you cannot place any more code after the return statement in that function.

IF STATEMENTS

Firstly, set a Boolean value to a variable,

is_male = True

If statement in python process the first line of code when the boolean value of the variable in the IF statement is True, i.e.

if is_male:

print (“You are a male”)

This would print “You are a male” to the screen, whereas if is_male = False, it’d do nothing.

if is_male:

print (“You are a male”)

else:

print (“You are not a male”)

Now, what about an IF statement that checks multiple boolean variables? e.g.

is_male = True

is_tall = True

if is_male or is_tall:

print “You’re either male or tall or both”

else:

print “You’re neither male nor tall”

an alternative to using **or** is to use **and** e.g.

if is_male and is_tall:

print “You are a tall male”

else:

print “You’re either not male or not tall or both”

Finally, by using the **elif** statement(s) between the **if** and **else** statements, we can execute a command or commands in the event that **is_male = True** but **is_tall is False**, i.e.

if is_male and is_tall:

print “You are male and tall”

elif is_male and not(is_tall):

print “You are not a tall male”

elif not(is_male) and is_tall:

print “You are tall but not male”

else:

print “You are neither male nor tall”

IF STATEMENTS AND COMPARISONS

The following examples show how you might compare numbers or strings using a function containing if statements and comparison operators.

```
#Comparison Operators
#Function to return the biggest number of three numbers
def max_num(num1, num2, num3):
if num1 >= num2 and num1 >= num3:
return num1
```

```
elif num2 >= num1 and num2 >= num3:
return num2
else:
return num3

print (max_num(3, 4, 5))

#Function to compare three strings
def match(str1, str2, str3):
if str1 == str2 and str1 == str3:
return "All strings match"
elif str1 == str2 and str1 != str3:
return "Only the first two match"
elif str1 != str2 and str2 == str3:
return "Only the second two match"
elif str1 == str3 and str1 != str2
return "Only the first and last match"
else:
return "None of them match"

print (match("Bob", "Alice", "Bob"))
```

python also supports <> as well as != and can also compare strings and numbers e.g. ’12’ <> 12

BUILDING A CALCULATOR

This calculator will be able to perform all basic arithmetic, addition, subtraction, multiplication and division.

```
#A Calculator
num1 = float(input("Enter first number: "))
op = input("Enter operator: ")
num2 = float(input("Enter first number: "))

if op == "+":
```

```
print(num1 + num2)
elif op == "-":
print(num1 - num2)
elif op == "/":
print(num1 / num2)
elif op == "*":
print(num1 * num2)
else:
print "Invalid operator"
```

DICTIONARIES

Key and Value pairs can be stored in python dictionaries. To create a dictionary to store say, three letter months and full month names, you'd use the following structure. Note that in a dictionary the keys must be unique.

```
monthConversions = {
"Jan": "January",
"Feb": "February",
"Mar": "March",
"Apr": "April",
"May": "May",
"Jun": "June",
"Aug": "August",
"Sep": "September",
"Oct": "October",
"Nov": "November",
"Dec": "December",
}
```

To retrieve the value for a given key, use **print(monthConversions[Sep])** or **print(monthConversions.get("Sep")** using the *get* function.

The get function also allows you to specify a default value in the event the key is not found in the dictionary, e.g.

print(monthConversions.get(“Bob”, “Key not in dictionary”)

WHILE LOOPS

The following example starts at 1 then loops until 10

```
#WHILE LOOP
i = 1
while i <= 10:
    print(i)
    i = i + 1)  #or use i += 1 to increment by 1
print ("Done with loop")
```

The while loop will execute the indented code while the condition remains True.

BUILDING A GUESSING GAME

```
#GUESSING GAME
secret_word = "cyberfella"
guess = ""
tries = 0
limit = 3
out_of_guesses = False

while guess != secret_word and not out_of_guesses: #will loop code while conditions are True
if tries < limit:
guess = input("Enter guess: ")
```

```
tries += 1
else:
out_of_guesses = True

if out_of_guesses:
#executes if condition/boolean variable is True
print ("Out of guesses, you lose!")
else:
#executes if boolean condition is False
print ("You win")
```

FOR LOOPS

Here are some examples of for loops

```
#FOR LOOPS
for eachletter in "Cyberfella Ltd":
    print (eachletter)

friends = ["Bob", "Alice", "Matt"]
for eachfriend in friends:
    print(eachfriend)

for index in range(10):
    print(index) #prints all numbers starting at 0 excluding
10

for index in range(3,10):
    print(index) #prints all numbers between 3 and 9 but not
10

for index in range (len(friends)):
    print(friends[index]) #prints out all friends at position
0, 1, 2 etc depending on the length of the list or tuple of
friends

for index in range (5):
    if index == 0:
```

```
        print ("first iteration of loop")
    else:
        print ("not first iteration")
```

EXPONENT FUNCTIONS

print (23)** #prints 2 to the power of 3

Create a function called raise_to_power to take a number and multiply it by itself a number of times,

```
def raise_to_power(base_num, pow_num):
result = 1
for index in range (pow_num):
#carry on multiplying the number by itself until you hit the range limit specified by pow_num
result = result * base_num
return result
```

2D LISTS AND NESTED LOOPS

In python, you can store values in a table, or 2D list, and print the values from certain parts of the table depending on their row and column positions. note that positions start at zero, not 1.

```
#Create a grid of numbers, that is 4 rows and 3 columns
number_grid = [
[1,2,3],
[4,5,6],
[7,8,9],
[0]
```

```
]
#return the value from first row (row 0) first column
(position 0)
print number_grid [0][0]
#returns 1

#return the value from third row (row 2) third column
(position 2)
print number_grid [0][0]
#returns 9


for eachrow in number_grid:
print (row)

for eachrow in number_grid:
for column in eachrow:
print (column)
#returns the value of each column in each row until it hits
the end
```

BUILD A TRANSLATOR

This little program is an example of nested if statements that take user input and translate any vowels in the string input to an upper or lowercase x

```
#CONVERTS ANY VOWELS TO A X
def translate(phrase):
translation = ""
for letter in phrase:
if letter.lower() in "aeiou":
if letter.isupper():
translation = translation + "X"
else
translation = translation + "x"
else:
```

```
translation =  translation + letter
return translation

print(translate(input("Enter a phrase: ")))
```

TRY EXCEPT

Catching errors in your program to prevent the program from being prevented from running. Error handling in python for example, if you prompt the user for numerical input and they provide alphanumerical input, the program would error and stop.

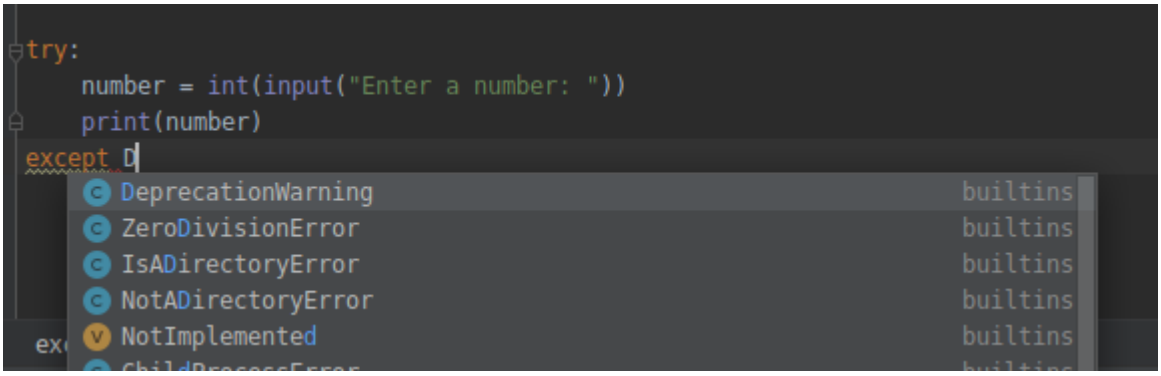
number = int(input(“Enter a number: “))

The variable number is set based upon the numerical, or more specifically the integer value of the users input. In order to handle all the potential pitfalls, we can create a **“try except”** block, whereby the code that could “go wrong” is indented after a **try:** and the code to execute in the event of an error, being indented after the **except:** block, e.g.

```
try:
number = int(input("Enter a number: "))
print(number)
except:
print("Invalid input, that number's not an integer")
```

Specific error types can be caught by specifying the type of error after except. Using an editor like pycharm will display the possible options or errors that can be caught but the specific error will be in the output of the script with it

stops.



So if we execute the code outside of a try: block, and enter a letter when asked for an integer, we'd get the following error output that we can then use to create our try: except: block to handle that specific *ValueError* error type in future.

```
Enter a number: f
Traceback (most recent call last):
  File "/home/matt/Software/Python/hello.py", line 268, in <module>
    number = int(input("Enter a number: "))
ValueError: invalid literal for int() with base 10: 'f'

Process finished with exit code 1
```

```
try:
    number = int(input("Enter a number: "))
    print(number)
except ValueError:
    print("Invalid input, that number entered is likely not an integer")
```

You can add multiple except: blocks in a try: except: block of code.

If you want to capture a certain type of error and then just display that error, rather than a custom message or execute

some alternative code then you can do this…

```
except ZeroDivisionError as err:
    print(err)
```

This can be useful during troubleshooting.

READING FROM FILES

You can “open” a file in different modes, read “r”, write “w”, append “a”, read and write “r+”

employee_file = open(“employees.txt”, “r”) #Opens a file named employees.txt in read mode

It’s worth checking that the file can be read, **print(employee_file.readable())**

You need to close the file once you’re done with it,

employee_file.close()

The examples below show different ways you can read from a file

```
#READING FROM FILES
employee_file = open("employees.txt", "r") #opens file read mode
```

```
print(employee_file.readable()) #checks file is readable
print(employee_file.read()) #reads entire file to screen
print(employee_file.readline()) #can be used multiple times to read next line in file
print(employee_file.readlines()) #reads each line into a list
print(employee_file.readlines()[1]) #reads into a list and displays item at position 1 in the list
employee_file.close() #closes file

for employee in employee_file.readlines():
    print (employee)
employee_file.close()
```

WRITING AND APPENDING TO FILES

You can append a a file by opening it in append mode, or overwrite/write a new file by opening it in write mode. You may need to add newline characters in append mode to avoid appending your new line onto the end of the existing last line.

```
#WRITING AND APPENDING TO FILE
employee_file = open("employees.txt", "a") #opens file append mode
employee_file.write("Toby - HR") #in append mode will append this onto the end of the last line, not after the last line
employee_file.write("\nToby - HR") #in append mode will add a newline char to the end of the last line, then write a new line

employee_file = open("employees.txt", "w") #opens file write mode
employee_file.write("Toby - HR") #in write mode, this will overwrite the file entirely

employee_file.close()
```

MODULES AND PIP

Besides the builtin modules in python, python comes with some additional modules that can be read in by your python code to increase the functionality available to you. This can reduce time since many things you may want to achieve have already been written in one of these modules. This is really what python is all about – the ability to pull in the modules you need, keep everything light and reduce time.

IMPORTING FUNCTIONS FROM EXTERNAL FILES

```
import useful_tools
print(useful_tools.roll_dice(10))
# MORE MODULES AT docs.python.org/3/py-modindex.html

SOME USEFUL COMMANDS
feet_in_mile = 5280
metres_in_kilometer = 1000
beatles=["John", "Ringo", "Paul", "George"]
def get_file_ext(filename):
    return filename[filename.index(".") + 1:]
def roll_dice(num):
    return random.randint(1, num)
```

To use a module in an external file, use **print(useful_tools.roll_dice(10))** for example (rolls a 10 sided dice).

EXTERNAL MODULES

Besides the additional internal modules that can be read into your python script, there are also many external modules

maintained by a huge community of python programmers. Many of these external modules can be installed using the builtin pip command that comes as part of python. e.g. **pip install python-docx** will install the external python module that allows you to read and write to Microsoft Word documents.

You can install pip using your package manager

```
matt@EliteDesk:/dev/disk/by-uuid$ sudo apt-get install python-pip
[sudo] password for matt:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  python-pip-whl
Recommended packages:
  build-essential python-all-dev python-setuptools python-wheel
The following NEW packages will be installed
  python-pip python-pip-whl
0 to upgrade, 2 to newly install, 0 to remove and 0 not to upgrade.
Need to get 1,803 kB of archives.
After this operation, 2,528 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://archive.ubuntu.com/ubuntu bionic-updates/universe amd64 python-pip-
whl all 9.0.1-2.3~ubuntu1.18.04.1 [1,653 kB]
Get:2 http://archive.ubuntu.com/ubuntu bionic-updates/universe amd64 python-pip
all 9.0.1-2.3~ubuntu1.18.04.1 [151 kB]
Fetched 1,803 kB in 1s (3,358 kB/s)
Selecting previously unselected package python-pip-whl.
(Reading database ... 247851 files and directories currently installed.)
Preparing to unpack .../python-pip-whl_9.0.1-2.3~ubuntu1.18.04.1_all.deb ...
Unpacking python-pip-whl (9.0.1-2.3~ubuntu1.18.04.1) ...
Selecting previously unselected package python-pip.
Preparing to unpack .../python-pip_9.0.1-2.3~ubuntu1.18.04.1_all.deb ...
Unpacking python-pip (9.0.1-2.3~ubuntu1.18.04.1) ...
Setting up python-pip-whl (9.0.1-2.3~ubuntu1.18.04.1) ...
Setting up python-pip (9.0.1-2.3~ubuntu1.18.04.1) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
matt@EliteDesk:/dev/disk/by-uuid$
```

To uninstall a python modules, use **pip uninstall python-docx** for example.

CLASSES AND OBJECTS

A class defines a datatype. In the example, we create a datatype or *class* for a Student.

```
class Student:
def __init__(self, name, major, gpa, is_on_probation):
self.name = name
self.major = major
self.gpa = gpa
self.is_on_probation = is_on_probation
```

This can be saved in its own class file called Student.py and can be imported into your python script using the command **from Student import Student** i.e. from the Student file, I want to import the Student class.

To create an object that is an instance of a class, or in our case, a student that is an instance of the Student class, we can use **student1 = Student("Bob", "IT", 3.7, False)**

print (student1.name) will print the name attribute of the student1 object.

BUILDING A MULTIPLE CHOICE QUIZ

If you're using pycharm to create your python code, then Click File, New, Python File and create an external class named *Question.py*. This will define the data type for the questions in your main multiple choice quiz code.

```
class Question:
def __init__(self, prompt, answer):
```

```
self.prompt = prompt
self.answer = answer
```

Now in your main code, read in that *Questions.py* class, create some questions in a list called *question_prompts*, and define the answers to those questions in another list called *questions*, e.g.

```
from Question import Question

question_prompts = [
    "What colour are apples?\n(a) Red/Green\n(b) Purple\n(c) Orange\n\n",
     "What colour are Bananas\n(a) Teal\n(b) Magenta\n(c) Yellow\n\n",
    "What colour are Strawberries?\n(a) Yellow\n(b) Red\n(c) Blue\n\n"
]

questions = [
    Question(question_prompts[0], "a"),
    Question(question_prompts[1], "c"),
    Question(question_prompts[2], "b"),
]
```

Now create a function to ask the questions, e.g.

```
def run_test(questions):
score = 0
for question in questions:
answer = input(question.prompt)
if answer == question.answer:
score +=1
print("You got " + str(score) + "/" + str(len(questions)) + " correct")
```

and lastly, create one line of code in the main section that runs the test.

run_test (questions)

OBJECT FUNCTIONS

Consider the following scenario – a python class that defines a Student data type, i.e.

```
class Student:
def __init__(selfself, name, major, gpa):
self.name = name
self.major = major
self.gpa = gpa
```

and some code as follows,

```
from Student import Student

student1 = Student("Oscar", "Accounting", 3.1)
student2 = Student("Phyllis", "Business", 3.8)
```

An object function is a function that exists within a class. In this example, we'll add a function that determines if the student is on the honours list, based upon their gpa being above 3.5

In the Student.py class, we'll add a *on_honours_list* function

```
class Student:
    def __init__(self, name, major, gpa):
        self.name = name
        self.major = major
        self.gpa = gpa

    def on_honours_list(self):
        if self.gpa >= 3.5:
            return True
        else:
            return False
```

and in our app, we'll add a line of code to check if a particular student is on the honours list.

```
from Student import Student

student1 = Student("Oscar", "Accounting", 3.1)
student2 = Student("Phyllis", "Business", 3.8)

print(student1.on_honours_list())
```

INHERITANCE

Classes can inherit the functions from other classes, this is done as follows. Consider a class that defines a Chef object, e.g.

```
class Chef:

def make_chicken(self):
print("The chef makes a chicken")

def make_salad(self):
print("The chef makes a salad")
```

```
def make_special_dish(self):
print("The chef makes bbq ribs")
```

Within the main app code, you can instruct the Chef as follows.

```
from Chef import Chef
myChef = Chef()
myChef.make_chicken()
myChef.make_special_dish()
```

But what if there was a Chinese Chef who could do everything that the Chef could do, but also made additional dishes and a different special dish? Creating an additional class for the ChineseChef as follows would facilitate this. e.g. *ChineseChef.py* would contain…

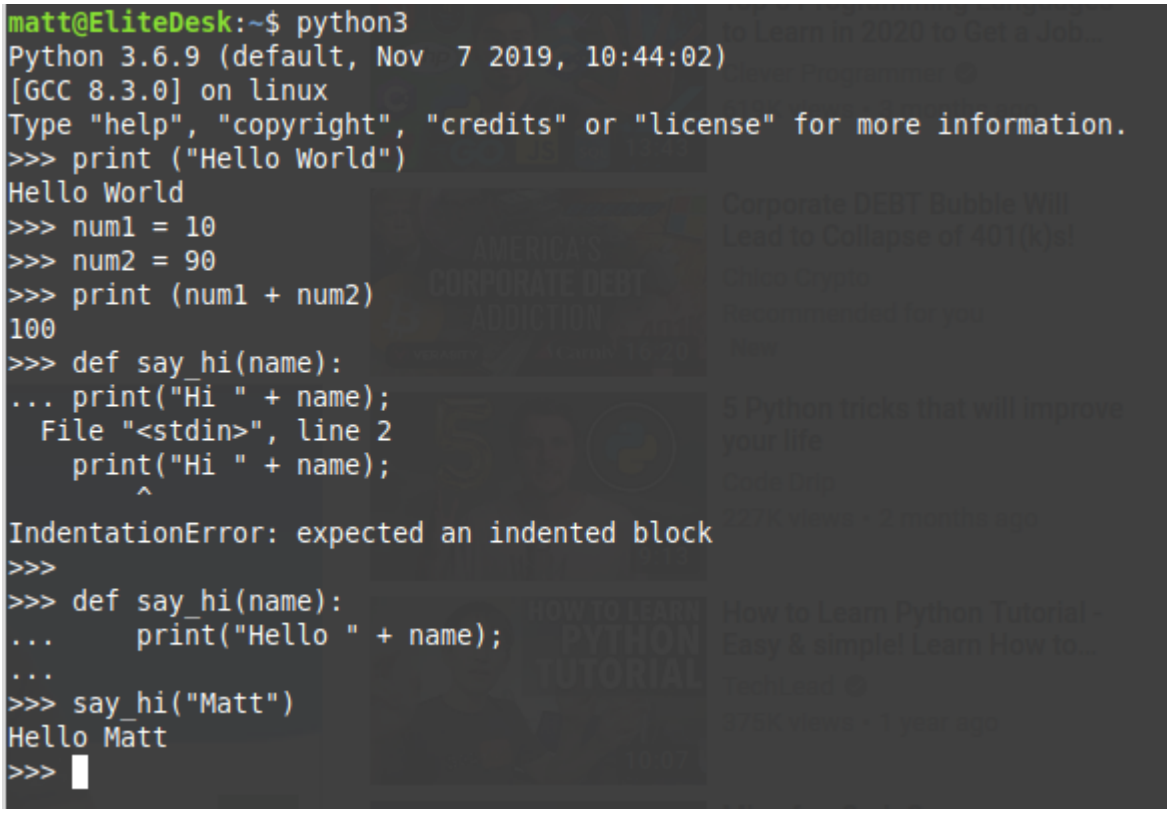
```
from Chef import Chef
class ChineseChef(Chef):
def make_special_dish(self):
print ("The chef makes Orange Chicken")

def make_fried_rice(self):
print ("The chef makes fried rice")
```

So, the class imports the other class, then the skills unique to the Chinese Chef are added, and also, any of the same skills overridden by re-defining them within the ChineseChef class.

PYTHON INTERPRETER

On Windows, Mac or Linux, you can access the python command line interpreter to perform some quick and dirty tests of your commands, Note that python is very particular about it's tab indented code where applicable – something that was done in say Shell Scripting at the discretion of the programmer for ease of readability – python really enforces it. e.g.



Using the python command will likely open an interpreter for python v2.x whereas the python3 command will open the interpreter for python v3.x. Be sure to add the path to the PATH environment variable if using Windows.

For coding in python, it's best to use a good text editor, Notepad++ (notepadqq on linux), or a proper coding text editor such as Visual Studio Code (runs on linux as well as Windows and Mac) or Atom or the best dedicated to writing python in particular is PyCharm. The community edition is free, or there is a paid for, Professional Edition.

I found PyCharm be be available via my Software Manager on Linux Mint.