Lesson 1 Exercises

Don’t just read these examples. Sit in front of your PC and do them and experiment.

# Section 1 – Getting Started

Start Python IDLE and at the interactive shell prompt type this piece of Python code:

>>> 2 + 2

Press the return/enter key and the result should be:

>>> 2 + 2

4

>>>

Good to see that Python can do addition!

# Numbers

Straight out of the box Python understands three ***types*** of numbers: integer, floating point, complex.

>>> type(2)

<class 'int'>

>>> type(2.0)

<class 'float'>

>>> type(4 + 5j) # (we will not be using complex numbers)

<class ‘complex’>

You should find they behave as you would expect.

# The Operators: +, -, \*, \*\*, /, //, %

Practice using these operators with a variety of integer numbers until you are comfortable with the results you are getting:

>>> 2 + 3 # addition

>>> 2 – 3 # subtraction

>>> 2 \* 3 # multiplication

>>> 2 / 3 # division

Now try them again but this time testing the type() of the result:

>>> type(2 + 3)

>>> type(2 - 3)

>>> type(2 \* 3)

>>> type(2 / 2)

>>> type(2 / 3)

The division operator ‘/’ always produces a ‘float’ result even when both numbers are integers and divide each other an exact number of times. This is a small but occasionally important difference.

A few more examples to try:

>>> 2 \* - 2 # ‘-‘ is the ‘unary negation operator’

>>> 2 \*\* 3 # exponentiation

>>> 11 / 2 # division you have already done

>>> 11 // 2 # do you see what this is doing?

>>> 11 % 2 # and this?

Try more examples using // and % until you understand what those operators do.

<https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex>

# BODMAS / BIDMAS / BEDMAS / PEMDAS etc

Brackets control the order in which operations are done:

>>> 2 - 3 + 4

>>> (2 - 3) + 4

>>> 2 - (3 + 4)

Try a few examples yourself with the other operators, for example:

>>> 11 // 2 \* 2 + 11 % 2

>>> 11 // (2 \* 2) + 11 % 2

>>> (11 // 2 \* 2 + 11) % 2

>>> (2\*\*41 + 2\*\*43)//1000%10 # can you explain result?

If the answers python gives you are not ‘obvious’ then try more examples until the results really are obvious.

# The Relational Operators: <, <=, ==, !=, >=, >

The relational operators are: <, <=, ==, !=, >=, >

In words: less-than, less than or equal, equal, not equal, greater than or equal, and greater than.

IMPORTANT: as well as pairs of numbers the relational operators can compare many other types of objects including strings (remember the quotation marks!)

Do try these:

>>> ‘A’ < ‘B’

>>> ‘A’ < ‘b’

>>> ‘a’ < ‘B’

>>> ‘a’ < ‘b’

Explain the results you get.

And now try these as well:

>>> 5 == 5

>>> 4 == 6

>>> 4 != 6

>>> 5 < 4 == 4 > 5

>>> 5 < 4 != 4 > 5

>>> ‘a’ < ‘b’ == ‘c’ < ‘z’

# Strings (text)

You can create a string by enclosing it in quotation marks.

>>> ‘this is some text in quotes’

‘this is some text in quotes’

>>> type(‘this is some text in quotes’)

<class ‘str’>

The operators + and \* also work on strings:

>>> ‘Monty’ + ‘Python’

>>> ‘MontyPython’

>>> ‘Spam’ + ‘ and eggs’ \* 5

‘Spam and eggs and eggs and eggs and eggs and eggs’

# Booleans

There are only two Boolean values, True and False.

>>> type(True)

<class 'bool'>

>>> type(False)

<class 'bool'>

Boolean values can be converted to numbers:

>>> int(True), int(False)

1, 0

Objects can be converted to Booleans using the built-in function bool(). Any non-zero or non-empty object becomes True, zero and empty object become False

>>> bool(1), bool(0)

True, False

>>> bool(‘any text’)

True

>>> bool(‘’) # empty string

False

Booleans and Boolean logic are studies in lessons 3 & 4.

# The Logical Operators: and, or , not

These are the operators of Boolean Logic which we will study later. For now, just try a few examples to see how they work.

>>> True and True

>>> True and False

>>> True or False

>>> False or False

>>> not True

>>> not False

>>> not 4 < 3

>>> not 4 < 3 and 5 < 4

# Section 2 – Values, Variables and Variable Names

Typing in literal values all the time is tedious. We need a way of being able to refer to a value by a name. We need variables.

Create a variable by giving it a name and assigning it a value:

>>> a = 3

>>> b = 5

All variable names must:

1. Begin with a letter or an underscore
2. Contain no spaces
3. Contain only letters, the digits 0 to 9, and underscores within them.

*A variable is the name of an address in memory that contains a value*

For example, here are some syntactically correct statements:

>>> bananas = 5

>>> my\_2nd\_FAV\_number = 1729 # Ramanujan’s taxi number

>>> false = True # syntactically correct but NOT recommended!

Assignment is NOT the same as arithmetic equality. This next assignment statement is perfectly legal in python but is obvious nonsense if interpreted mathematically:

>>> a = a + 3

The meaning of the statement in Python is:

1. Get the value from the memory location that the variable name ‘a’ points to
2. Add 3 to that value
3. Store the result of that calculation in the memory location the variable name ‘a’ points to, thus overwriting and erasing the previous value in that memory location.

We can use variables in expressions to create values which we can then use directly or assign to new or existing variables.

>>> c = a + b

>>> c\*\*2 – b\*\*2 – a\*\*2

70

When writing programs do give your variables ‘meaningful names’. In other words, a name that gives the reader (you) a hint as to what the value it refers to represents:

>>> weight\_in\_kg = 729 # weight in kilograms

>>> weight\_in\_pnds = weight\_in\_kg \* 2.2 # same wight in lbs

You get the idea.

For ‘throw away’ and practice snippets of code it doesn’t matter but for anything you will keep and use again, using meaningful names will save you lots of time in your future coding life. Even though you wrote the code, and tested it, and got it to work, when you come back to your code in 3 months’ time (or 3 days!) you will struggle to remember which variables refer to what, and hence what your code does, or what is wrong with it:

r1 = -b-s(b\*\*2-4\*a\*c)/2\*a # calculate r1 (!???)

And that’s an easy one!

# Saving and Re-using your work

If you had to type in the code every time you wanted to execute a program it that would be very tedious to say the least! Fortunately, we can save the code we write, and then reload and execute it later (phew!)

## Organise your file-space

Use File Manager (Windows) or Finder (macOS) to create a folder for this course. Perhaps call ‘coding’ or ‘lin6209’ or whatever is meaningful to you.

Within that course folder create subfolders for each unit of work: week1, week2, etc is easy and obvious. IMPORTANT: do not leave any spaces in the names of these subfolders, use the same naming convention as for Python variable names.

Same graphics as used for PPT slides:

1. Create script
2. Save script into your chosen folder
3. Run script

Re-using your Python program:

1. Open the previously saved Python script in IDLE
2. Press F5 or ‘Run’ from the menu to run it

# More about Strings

Remember this: ***Strings are immutable***. Tuck that fact away for now and we will come back to the implications of what it means later.

## Creating Strings

You can create a string in python by using quotes:

>>> ‘a’

‘a’

>>> type(‘a’)

<class ‘str’>

Single quotes and double quotes have the same effect:

>>> type(“a”)

<class ‘str’>

Several of the operators we have just met also work on strings:

>>> ‘a’ + ‘a’

‘aa’

>>> ‘a’ \* 5

‘aaaaa’

It does not matter if you use double or single quotes – AS LONG AS THEY MATCH:

>>> ‘apple’ == “apple”

True

>>> 'apple"=="apple'

'apple"=="apple'

Can you explain the difference? Experiment to prove to yourself you really do understand.

## The length of a string

The built-in function len() answers the number of characters in a string:

>>> len('apple pie’)

9

## String Indexing

Create a few string variables to practice with:

>>> a = ‘apple’

>>> b = ‘banana’

>>> c = ‘cherry’

The individual characters of a string can be referenced by their index number.

Remembering that computer scientists count from 0:

>>> a[0]

‘a’

>>> a[1]

‘p’

>>> a[4]

‘e’

Do not go past the last index!

>>> a[5]

IndexError message

You can go negative

>>> a[-1]

‘e’

>>> a[-4]

‘p’

>>> a[-5]

‘a’

But once again, do not go off the end (beginning!)

>>> a[-6]

## another IndexError message

## String Slicing

Taking the logic of indexing a step further, we can slice whole pieces out of the string:

>>> a[2:4]

‘pl’

>>> a[1:-1]

‘ppl’

## and reassemble the parts using string operators

>>> d = a[-2:] + a[:2]

>>> d

‘leap’

We can also use different sized steps to move through the string:

>>> a='abracadabra'

>>> b=a[0:-1:2]

>>> b

'arcdb'

Practice with your own examples until you are sure you understand.

## String Assignment

You might think you can do this:

>>> a[2] = ‘e’

or:

>>> a[2] = a[4]

But you can’t. Try it and see. Why?

Hint: Read the error message. What is it trying to tell you?

Hint: Re-read the first sentence of this ‘Strings’ section and then the error message Python gave.

## String Methods

Every string in python is an object which means it contains both data and code.

The code associated with an object is called its ‘methods’.

We can call these methods using the template:

>>> object\_name.method\_name(parameters)

Strings have lots of methods. See the Python documentation at page:

<https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str>

A few examples should suffice to get you going under your own steam:

>>> a = ‘abracadabra

>>> a.upper()

‘ABRACADABRA’

>>> a.count(‘b’)

2

>>> a.isalpha()

True

>>> a.isdigit()

False

>>> a.replace(‘a’, ‘z’)

'zbrzczdzbrz'

>>> a.split(‘b’)

['a', 'racada', 'ra']

>>> a.title().swapcase()

'aBRACADABRA'

>>> a.upper()

'ABRACADABRA'

Check what is happening by reading the documentation. Try a few other methods yourself.

## Converting strings ↔ integers ↔ floats

Python has some handy built-in functions that convert between numbers and strings:

>>> int(‘5’)

5

>>> float(‘5’)

5.0 # notice the decimal point

>>> str(5)

‘5’

>>> str(float(int(‘5’)))

‘5.0’

# The input() and print() functions

These two built-in functions are used for getting text input from the user and sending the output

from programs to the console as text.

## The built-in function print()

Place whatever variables or values you want printed to the screen between the brackets, separated by commas if there is are more than one.

The statement:

print(any\_object)

Will send a textual representation of any\_object to the standard output which is by default set to be the screen.

Executing a module containing the statement:

print(‘spam’)

Results in the output:

spam

Executing:

spam = ‘spam’

and = ‘and’

space = ‘ ‘

print(spam, and, spam)

print(spam + space + and + space + spam)

print(spam.capitalize(), and, spam)

Outputs:

spam and spam

spam and spam

Spam and spam

When you really do need the same quote symbol inside a string as defines it, use the e*scape character* ‘\’

Print('apple\’s’)

apple’s

print('both \' and " in this string'

both \' and " in this string

The backslash *escape symbol* \ can also print tabs and newlines and other ‘special’ characters. Consult the print() function documentation to learn more.

These examples all use strings. Numbers and any other objects you choose work as well, even the functions print() and input().

print(print, input) # note: just the names, not the brackets

We will explain the result next lesson:

## The built-in function input()

The input() function pauses execution of the program and waits for the user to type something and then press enter.

Executing:

name = input(‘Hello, what is your name:’)

print(‘Pleased to meet you’, name)

Results in the output:

Hello, what is your name:

Peter # user types this and presses ‘enter’ key

Pleased to meet you Peter

IMPORTANT: The input received from an input() statement is ALWAYS a string (text)

Executing:

num1 = input(‘Number 1 =’)

num2 = input(‘Number 2 =’)

num3 = num1 + num2

print(num1, ‘+’, num2, ‘=’, num3)

Results in the output:

Number 1 =

2

Number 2 =

3

2 + 3 = 23

The reason for this is that the addition is adding (concatenating) two strings, not adding two numbers. To get the arithmetic sum we need to use the built-in function int() or float() to convert the string to a number

num1 = input(‘Number 1 =’)

num2 = input(‘Number 2 =’)

num3 = int(num1) + int(num2)

print(num1, ‘+’, num2, ‘=’, num3)

With the same inputs this will output:

2 + 3 = 5

# The Built-in functions

<https://docs.python.org/3/library/functions.html>

You have already used several of the built-in functions

* int()
* float()
* str()
* type()
* input()
* print()
* len()

There are many other built-in function and more than a few are quite technical in their nature.

Ones that WILL be useful to you in the short term include:

* ascii()
* bool()
* chr()
* format() # Too much detail for now, just know that it ‘formats’ strings
* help()
* id()
* max()
* min()
* ord()
* reversed() # useful but tricky to use. Try this: >>> '\_'.join(reversed('spam'))
* sorted() # at least result print()’s. Try this: >>> ''.join(sorted('spam'))
* vars() # a list of the variables in current environment

No need to study these in detail. Just try a simple example or two so you are confident you can invoke it and see it working with an example or two. That should be enough to ensure that in future you will ‘know’ that Python has that particular nugget of functionality ‘built in’.