Python, a very gentle introduction

Release 2023.11.13 - 2023.11.17

Douglas Macdonald

Oct 12, 2023

CONTENTS

CHAPTER

ONE

CLASSES 1 - A FIRST LOOK AT *CLASSES* [CLASS-01-METHODS.RST]

- Like functions, **classes** are a way of grouping and organising your code.
- Classes are associated with a programming style called *object oriented* programming.
- You will find out more about classes and there use in the following exercises.

1.1 first_class.py

Write and test the following code, `first_class.py`.

```
class FirstClass:
    def method(self):
        print("Hello, from the class.")

instance_of_class = FirstClass()
instance_of_class.method()
```

1.2 first_class_ott_comments.py

Make a copy of `first_class.py` code and save it as `first_class_ott_comments.py` and add a comment to each line explaining what it does.

1.3 class_new_method.py

Make a copy of `first_class.py` code and save it as `class_new_method.py`. Add and test your own new method.

CHAPTER

TWO

CLASSES - RETURN FROM METHODS [CLASS-02-RETURN.RST]

Similar to functions, an objects method can return values.

2.1 method_return.py

Write this program, test, experiment and explain in comments what is going on.

```
class ClassWithReturnMethod:

    def get_double_hi(self):
        return 2 * "Hello "

instance_of_class = ClassWithReturnMethod()
val = instance_of_class.get_double_hi()
print(val)
```

CHAPTER

THREE

CLASSES - METHODS WITH PARAMETERS [CLASS-03-PARAMS.RST]

In a similar manner to functions, methods can take arguments.

3.1 method_parameter.py

```
class ClassMethodWithParameter:
    def times(self, val_1, val_2):
        print(val_1 * val_2)

    def add(self, val_1, val_2):
        print(val_1 + val_2)

instance = ClassMethodWithParameter()
instance.times(3, "hi ")
instance.add(1.2, 2)
```

CHAPTER

FOUR

CLASSES __INIT__ OPERATOR OVERLOADING [CLASS-04-OVERLOAD.RST]

Particular combinations of underscores at the beginning and/or end of names in your program can have special meanings. The double underscore at the beginning and end of a class method name (**operator overload methods**) are typically reserved for built-in methods or variables. `__init__` is used in a class to initiate an object.

4.1 initialisation_of_class.py

Write and test the following code, `initialisation_of_class.py`.

```
class InitClass:
    def __init__(self):
        print ("Instance")
    def method(self):
        print("Like a function.")

instance_1 = InitClass()
instance_1.method()
instance_1.method()

instance_2 = InitClass()
instance_2.method()
instance_2.method()
```

4.2 initialisation_of_class_ott_comments.py

Make a copy of your previous code and save it as `initialisation_of_class_ott_comments.py` and add a comment to each line explaining what it does.

4.3 double_class.py

Classes - updating internal attribute

Write and test this program which uses a method to update an attribute.

```
class DoubleClass:
    def __init__(self, val_in):
        self.value = val_in

    def double_value(self):
        self.value = 2 * self.value


instance_of_double_class = DoubleClass(1)
print(instance_of_double_class.value)

instance_of_double_class.double_value()
print(instance_of_double_class.value)

instance_of_double_class.double_value()
print(instance_of_double_class.value)

instance_of_double_class = DoubleClass("Hello ")
instance_of_double_class.double_value()
print(instance_of_double_class.value)
```

4.4 double_class_ott_comments.py

To demonstrate your understanding, fully comment the code then save as `double_class_ott_comments.py`.

CHAPTER

FIVE

CLASSES - OPERATOR OVERLOADING __ADD__ AND __SUB__ [CLASS-05-OVERLOAD-ADD_SUB.RST]

If appropriate, you can make the + and - operators work with your objects with the methods

> `__add__` makes the + work.
>
> `__sub__` makes the - work.

5.1 celsius.py

Write and test this program `celsius.py`.

```
# `celsius.py`

class Celsius:
    def __init__(self, temp_in):
        self.temp = temp_in
    def add(self, rhs):
        new_temp_value = self.temp + rhs.temp
        return Celsius(new_temp_value)

temp_1 = Celsius(32)
temp_2 = Celsius(100)
temp_3 = temp_1.add(temp_2)
temp_3.temp
print(temp_1.temp, temp_2.temp, temp_3.temp)
```

5.2 celsius_overload_01.py

Write and test this program `celsius_overload_01.py`. This replaces the add method of `celsius.py` with `__add__`.

```
# celsius_overload_01.py

class Celsius:
    def __init__(self, temp_in):
        self.temp = temp_in
    def __add__(self, rhs):
        new_temp_value = self.temp + rhs.temp
```

(continues on next page)

(continued from previous page)

```
        return Celsius(new_temp_value)
    def subtract(self, rhs):
        new_temp_value = self.temp - rhs.temp
        return Celsius(new_temp_value)

temp_1 = Celsius(32)
temp_2 = Celsius(100)
temp_3 = temp_1 + temp_2
print(temp_1.temp, temp_2.temp, temp_3.temp)
```

5.3 celsius_overload_02.py

Change your `celsius_overload_01.py` program so that the following `celsius_overload_02.py` will work.

```
# celsius_overload_02.py
temp_1 = Celsius(32)
temp_2 = Celsius(100)
temp_3 = temp_1 + temp_2 - temp_2 - temp_2
print(temp_1.temp, temp_2.temp, temp_3.temp)
```

CHAPTER

SIX

CLASSES 4 - OPERATOR OVERLOADING __STR__ [CLASS-06-OVERLOAD-STR.RST]

`__str__` returns user friendly string. The built-in functions `str()` and `print` look for this method.

6.1 `celsius_overload_03.py`

Now experiment with the `__str__` overload method. Test your class with `celsius_overload_03.py` and the `print` statement.

```
class Celsius:
    ...

    def __str__(self):
        return str(self.temp)

    ...
...

print temp_1, temp_2, temp_3
```

> python celsius_overload_03.py

32 Celsius 100 Celsius -68 Celsius

6.2 `celsius_ott_cmts.py`

Make a copy of your final version of `celsius_overload_03.py`, fully comment and save it as `celsius_ott_cmts.py`.

CHAPTER

SEVEN

CLASSES 5 - INHERITANCE [CLASS-07-INHERITANCE.RST]

Inheritance can be a useful mechanism for reusing code but it can easily become complex. Indeed, you will not often need inheritance and multiple inheritance (not covered here) almost never. In my opinion, it is often easier to use composition (using classes, objects and modules directly inside a class).

7.1 parent_and_child.py

Try to predict what the this code will do before testing in.

```
# parent_and_child.py

class ParentClass:
    def message(self):
        print('Parent method')

class ChildClass(ParentClass):
    def __init__(self, num):
        self.number = num

child_object = ChildClass(10)
child_object.message()
```

7.2 simple_inheritance.py

Here is a small inheritance exercise for you to try, `simple_inheritance.py`.

```
# simple_inheritance.py

class Person():
    def __init__(self, name, age, pob):
        self.name = name
        self.age = age
        self.pob = pob

    def print_info(self):
        print("Name:", self.name)
        print("Age:", self.age)
        print("Place of birth:", self.pob)
```

(continues on next page)

(continued from previous page)

```
class Student(Person):
    def __init__(self, name, age, pob, id_1):
        super(Student, self).__init__(name, age, pob)
        self.id = id_1

    def print_info(self):
        super(Student, self).print_info()
        print("Student ID:", self.id)


s = Student("David", 21, "Glasgow", 1234)
s.print_info()
```

CHAPTER

EIGHT

[CLASS-08-STR.RST]

```
def slice_strings(s, index_list):
    """Apply the slicing given slice positions in the index_list."""
    return [s[i:j] for i,j in zip(index_list[:-1], index_list[1:])]


def slice_on_count(s, count_for_letters):
    """Slice by the counts for each bin."""

    index_list = cumsum(count_for_letters)
    return slice_strings(s, index_list)


def equal_chunk_counts(s, n):
    """List of size of a letters chunks all with same number."""

    size_of_chunk = len(s) // n
    count_for_letters = [size_of_chunk] * n

    return count_for_letters


def cumsum(list_):
    """Cumulative sum"""

    cum_list = [0]
    sum_ = 0
    for i in list_:
        sum_+=i
        cum_list.append(sum_)

    return cum_list


def remainder_mask(s, n):
    """1s in remainder position and 0s in non-remainder position."""
    remainder = len(s) % n
    return ([1] * remainder) + ([0] * (len(s) - remainder))


def add_lists(list_1, list_2):
```

(continues on next page)

(continued from previous page)

```
    """Add by index position."""
    return [i+j for i,j in zip(list_1, list_2)]


def floor_div(s, n):
    """String floor division function."""
    count_for_letters = equal_chunk_counts(s, n)
    return slice_on_count(s, count_for_letters)


def div(s, n):
    """String division fumction."""
    count_for_letters = equal_chunk_counts(s, n)
    remainder_mask_ = remainder_mask(s, n)
    count_for_letters = add_lists(count_for_letters, remainder_mask_)
    return slice_on_count(s, count_for_letters)


def divmod_str(s, n):
    """Return the tuple (div_chunks, remainder_chunk)"""
    count_for_letters = equal_chunk_counts(s, n)
    count_for_letters.append(len(s)%n)
    chunks_with_remainder = slice_on_count(s, count_for_letters)
    return chunks_with_remainder[0:-1], chunks_with_remainder[-1]


def mod_str(s, n):
    return divmod_str(s, n)[-1]


class MyStr:

    #def __new__(cls, *args, **kw):
    #     return str.__new__(cls, *args, **kw)
    #

    def __init__(self, s):
        self.s = s

    def __truediv__(self, n):
        return div(self.s, n)

    def __floordiv__(self, n):
        return floor_div(self.s, n)
```

```
s = MyStr('12345678s91')
```

```
s/3
```

```
s//3
```

```
s = "123456789101234"
divmod_str(s, 3)
```

```
mod_str(s, 3)
```

CHAPTER

NINE

ERRORS 2 - TRY AND EXCEPT [EXCEPT-01-TRY.RST]

Errors detected during program execution are called *exceptions*. Exceptions can be triggered automatically on errors, or manually in your code. They can be intercepted and acted upon by your code and allow your program to continue. However, especially during development, on an error you will want the program to stop. In short, exceptions can be used, especially in an **emergency**, to abandon all the current activity and allow for jumps out of large chunks of code.

Given an exception, `try/except` can be used to handle and allow your code to continue. The general `try/except` syntax is:

```
try:
    pass
    # primary action
except:
    pass
    # run if any exception raised
```

Practice error handling with the following exercise.

9.1 broken.py

This is a broken piece of code. Write, run and make a note of the error message.

```
print(int(2.1))
print(int('2'))
print(int('cat'))
```

9.2 handled.py

Here, instead of fixing, in this version of the previous code the error is handled. Write, test and experiment with this code, `handled.py`.

```
print(int(2.1))
print(int('2'))

try:
    print(int('cat'))
except ValueError:
    print('cat is not a number')
```

CHAPTER

TEN

ERRORS 3 - ERROR HANDLING AND INPUT [EXCEPT-02-INPUT.RST]

10.1 error_handling.py

Using `input()` could crash if you get unexpected input. Write and test this program, `error_handling.py`. Explain the key word `continue` and error handling `try/except` code works in your comments.

```
while(True):

    user_input = input(
        "Input an integer number 1 to 10 ('q' to quit): ")

    if user_input == 'q':
        break

    try:
        user_input = int(user_input)
    except:
        print("Boo! That is not an integer.")
        continue

    if (1 <= user_input <= 10):
        print("Great! That's a valid number.")
    else:
        print(
            "Boo! That integer is not in the range 1 to 10.")
```

CHAPTER

ELEVEN

ERRORS - RAISE [EXCEPT-03-RAISE.RST]

As well as Python giving errors automatically, you can also signal errors using `raise`.

Write and test the following programs and explain in comments what is going on.

11.1 raise_01.py

```
while True:
    try:
        x = int(input("Give me a number: "))
        break
    except ValueError:
        print("Try again.")
```

CHAPTER

TWELVE

ERRORS - RAISE 2 [EXCEPT-033-RAISE.RST]

12.1 raise_02.py

```
while True:
    try:
        x = int(input("Give me a number 1-10: "))
        if not (1 <= x <= 10):
            raise TypeError("Not between 1 and 10, inclusive.")
        break
    except (ValueError, TypeError) as  e:
        print("Try again.", e)
```

12.2 raise_03.py

The previous code was for illustration. More correctly, remove the `TypeError` and its handling but leave the message. Test your code.

CHAPTER

THIRTEEN

ERRORS - ASSERT [EXCEPT-04-ASSERT.RST]

assert conditionally raises and exception - often used in debugging and development.

```
>>> assert True, "Assert error"


>>> assert False, "Assert error"
```

CHAPTER

FOURTEEN

ERRORS 4 - TRY, EXCEPT, ELSE AND FINALLY [EXCEPT-05-ELSE_FINALLY.RST]

There are four statements which are concerned with exceptions and exception handling:

```
try:
    # primary action
    pass
except:
    # if any exception raised
    pass
else:
    # if no exception raised
    pass
finally:
    # Always perform these clean up actions
    # whether an exception occurs or not.
    pass
```

14.1 finally_1.py

What is the result of this simple error handling?

```
# finally_1.py

try:
    1/0
except:
    msg = 'This message.'
finally:
    msg = 'That message.'

print("What message: ", msg)
```

CHAPTER

FIFTEEN

FILES 1 - INTRODUCTION [FILE-01-READ.RST]

Like most languages, Python can read data from files.

15.1 line_numbers.txt

With a text editor, create the file `line_numbers.txt`, with the following content:

```
Line one
Line two
Line three
Line four
Line five
Line six
```

15.2 file_reader.py

Write a program `file_reader.py` to read the text from the `line_numbers.txt` file you created. The `r` mode is to tell your operating system that you are opening the file for **r**eading.

```
file_name = "line_numbers.txt"

mode = 'r'
f = open(file_name, mode)

text = f.read()
f.close()

print(text)
```

15.3 file_high_read.py

Using the following text file, high.txt:

```
Python is a high-level
programming language.
A program is a sequence
of instructions.
```

- Create a program, file_high_reader.py, to read in the file high.txt and display the contents to the console. Make sure you close the file once you have finished with it.

15.4 file_readline.py

- Write this program to read in one line of high.txt and display it to the console.

```
f = open('high.txt')
line = f.readline()
print(line)
f.close()
```

CHAPTER

SIXTEEN

FILE READ, FOR AND READLINES [FILE-012-READ-FOR-READLINES.RST]

Use the following text file, high.txt.

```
Python is a high-level
programming language.
A program is a sequence
of instructions.
```

16.1 file_for.py

- Using a for loop, read in and print out a file line by line.

```
# file_for.py

f = open('high.txt')
for line in f:
    print(line)
f.close()
```

16.2 file_list.py

- Write a program to read each line of text2.txt into a list.

```
# file_list.py

f = open('high.txt')
contents = f.readlines()
print(contents)
f.close()
```

16.3 Files - readline method

Use your file, line_numbers.txt containing:

```
Line one
Line two
Line three
Line four
Line five
Line six
```

16.4 read_two_lines.py

Write, test and comment this program, read_two_lines.py, to display two lines of line_numbers.txt.

```
# read_two_lines.py

file_name = 'line_numbers.txt'
linenumbers_file = open(file_name)
line = linenumbers_file.readline()
print(line)
line = linenumbers_file.readline()
print(line)
linenumbers_file.close()
```

16.5 read_line.py

Write this short program, read_line.py, to display the first line of line_numbers.txt.

```
# read_line.py

file_name = 'line_numbers.txt'
linenumbers_file = open(file_name)
line = linenumbers_file.readline()
linenumbers_file.close()
print(line)
```

16.6 Files - for and swapcase method

Use file, line_numbers.txt, with the following content:

```
Line one
Line two
Line three
Line four
Line five
Line six
```

16.7 swapcase_lines.py

You can also use `readline` with a `for` loop. Try it out with `swapcase_lines.py`.

```
# swapcase_lines.py

file_name = 'line_numbers.txt'
linenumbers_file = open(file_name)
for line in linenumbers_file:
    print(line.swapcase())
linenumbers_file.close()
```

16.8 swapcase_lines_ott_comments.py

Copy `swapcase_lines.py`, thoroughly comment and save as `swapcase_lines_ott_comments.py`.

CHAPTER

SEVENTEEN

FILES 2 - WRITING 1 [FILE-03-WRITE.RST]

We have previously looked at *reading* files. To do the opposite and *write* to a file, you have to open it in write mode. This is done by setting the second parameter of the function `open` to `'w'` write mode - `open('file_name','w')`.

17.1 `file_writer.py`

Write and test this program, `file_writer.py`. It saves a message to a file `text.txt`.

```
file_name = "text.txt"
mode = 'w'
f = open(file_name, mode)

text = "Hi, from the test file."
f.write(text)

f.close()
```

17.2 Check

After running the program, check it worked by opening `text.txt` with your editor.

```
Hi, from the test file.
```

17.3 `writelines`

The file `writelines` method writes all the strings in a list into a file.

17.4 file_writelist.py

Write and comment this program `file_writelist.py` that uses `writelines` to save the entries of a *list* to a file.

```
num_list = ['0\n','1\n','2\n','3\n','4\n','5\n','6\n','7\n','8\n','9\n']

file_name = "data.dat"
mode = 'w'

f = open(file_name, mode)
f.writelines(num_list)

f.close()
```

17.5 data.dat

Use your editor to open the file you created, `data.dat`, and check your previous program worked correctly.

```
0
1
2
3
4
5
6
7
8
9
```

17.6 file_readlist.py

Write a program `file_readlist.py` to read your list file and compare it against the original.

17.7 file_writelist_2.py

Write a program, `file_writelist_2.py`, that saves a list. Check that the file `data_2.dat` was created correctly. Write a program to read in the previous **data.dat** into a list of integers.

```
file_name = "data_2.dat"
mode = 'w'
f = open(file_name,mode)

num_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

for item in num_list:
    f.write(str(item) + '\n')

f.close()
```

17.8 file_hello_writer.py

Write a program, file_hello_writer.py, that writes a file, hello.txt, containing the message, Hi, from the test file..

```
file_name = 'hello.txt'
mode = 'w'
f = open(file_name, mode)

text = "Hi, from the test file."
f.write(text)

f.close()
```

17.9 file_sat_read_write.py

Create a program, file_sat_read_write.py, to write text from a file saturday_in.txt (below) into the file saturday_out.txt. Make sure you close the file once you have finished with it.

17.10 saturday_in.txt

```
Well at least
it is
Saturday.
```

17.11 file_sat_read_write.py

```
f_in = open('saturday_in.txt')
contents = f_in.read()
f_in.close()

f_out = open('saturday_out.txt', 'w')
f_out.write(contents)
f_out.close()
```

CHAPTER

EIGHTEEN

[FILES-04.RST]

18.1 files

- Create and open a file ready for writing.
- Open a file ready for ready reading.
- Open a file to append to the end.
- Open a file for reading and writing
- Read in 10 characters.
- Move to 20th characters.
- Flush the data to the file.

CHAPTER

NINETEEN

FILES 5 - `WITH` STATEMENT AND CONTEXT MANAGERS [FILE-05-WITH.RST]

In Python, the `with` statement is used to carry out automatically **setup** and **teardown** code.

For files, `with` will automatically call the `close` method for you.

Write and test these two programs.

```
# hello_without.py
file = open("without.txt", "w")
file.write("Hello, without!")
# If something goes wrong during the write,
# then the program might crash before the
# close is properly called. Moreover, for efficiency,
# you are also not guaranteed that the system's
# write is completed until the `close` is called.
file.close()
```

The `with` statement can solve these problems.

```
# hello_with.py
# Open the file using the `with` statement.
with open("hello_with.txt", mode="w") as file:
    # The file is open inside the block.
    file.write("Hello, with!")
    # The file is still open.

# Leaving the `with` block, the file is automatically closed.
```

CHAPTER

TWENTY

FUNCTIONS 2 - INDENTATION AND YOUR OWN FUNCTIONS [FUNC-01-SIMPLE.RST]

In Python, indentation of text from the left is significant. It is used to group and indicate blocks of code. By convention 4 spaces are used.

20.1 Note

- If you are using a simple text editor, change the tab-key configuration to indent as 4 spaces.
- Although **tabs** (tab characters) can be used instead of spaces, it is still better to use spaces.
- Either way, be consistent within a file.

20.2 `first_function.py`

Write and test the following program, `first_function.py`.

```
def first_function():
    print("Hello, from first_function.")
    print(5 * 4)
    print("Bye, from first_function.")

print("Outside function.")
first_function()
first_function()
print("End of program.")
```

20.3 `ott_comments_first_function.py`

Copy the previous code, `first_function.py` , into a new program `ott_comments_first_function.py` . Write a comment above each line explaining what it does.

20.4 `three_times_first_function.py`

Copy the `first_function.py` code into a new program, `three_times_first_function.py`. Instead of two, call the function three times. Test and write a comment above each line explaining what is going on.

20.5 `my_function_twice.py`

Write a program, `my_function_twice.py`, containing a function, `my_function()`.

This function should print a friendly message. In your program, call this function twice.

20.6 `doubles.py`

Create a program, `doubles.py`, with the following specification:

- It should contain one function.
- It should display the result of `2 * "double "`.
- Call your function 3 times.

The output should be:

```
double double
double double
double double
```

CHAPTER

TWENTYONE

ERRORS - PASS [FUNC-02-PASS.RST]

`pass` is a place holder statement that does nothing other than tell Python that you meant to do nothing. It can be useful when you are developing a program.

21.1 stubs_error.py

Write and test this program, `stubs_error.py`, and note the output.

```
def func_1():

def func_2():

def func_3():

func_1()
func_2()
func_3()
```

21.2 stubs_fixed.py

Using `pass` fix the previous program and save as `stubs_fixed.py`. Check that it now runs without an error.

```
def func_1():
    pass

def func_2():
    pass

def func_3():
    pass

func_1()
func_2()
func_3()
```

In this way you can use *function stubs* to help plan your code.

CHAPTER

TWENTYTWO

FUNCTIONS - RETURN [FUNC-03-RETURN.RST]

22.1 second_function.py

Write and test the following program, second_function.py.

```
def second_function():
    val = 4 + 5
    return val

returned_value = second_function()
print(returned_value)
```

22.2 second_func_ott_comments.py

Write a well commented version of **second_function.py** and save it as **second_func_ott_comments.py**.

22.3 doubles_55.py

Write a program, **doubles.py**, that doubles 55 and returns the result. Display the returned result.

CHAPTER

TWENTYTHREE

FUNCTIONS - RECURSION [FUNC-04-RECURSION.RST]

A **recursive** function is a function that calls itself from within itself. The process is called **recursion**.

23.1 recursive_function.py

Write and test this recursive function.

```
def recursive_function(n):

    print(n)
    if n >= 1: #recursive condition
        n = n - 1
        return recursive_function(n)
    else:
        return(n) #end

print(recursive_function(-1))
print(recursive_function(4))
print(recursive_function(4.1))
```

23.2 recursive_function_ott_comments.py

Make a copy of the previous code and thoroughly comment.

CHAPTER

TWENTYFOUR

FUNCTIONS - PARAMETERS [FUNC-05-PARAM.RST]

As well as providing return values, functions can also take inputs.

24.1 double.py

What is the output from these two calls to `double`?

```
def double(var):
    return 2 * var

print("double(2) =", double(2))
print("double('python ') =", double('python '))
```

24.2 even.py

Write and call a function `is_even` that that takes one argument and returns `True` if the input value is even otherwise `False`. Hint `%` operator.

24.3 multiply_three.py and iterative development

It is a good idea to build these up in stages, testing as you go. Write and test the following program, `multiply_three.py`. Write small sections and test as you go along as follows.

24.4 multiply_three.py

Write and test. If you do not already know, ask the internet what `pass` does.

```
def multiply_three_parameters():
    pass

print(multiply_three_parameters())
```

24.5 multiply_three.py

Write and test.

```
def multiply_three_parameters(a):
    return a

print(multiply_three_parameters(1))
```

24.6 multiply_three.py

Continue your writing and testing until you have completed this program.

```
def multiply_three_parameters(a, b, c):
    x = 2 * a
    y = 2 * b
    z = 2 * c
    return x, y, z

print(multiply_three_parameters(1, 2, 3))
print(multiply_three_parameters(1.0, 2.2, 3.3))
print(multiply_three_parameters('hello', 2.2, 3.3))
d, e, f = multiply_three_parameters('hello', 2.2, 'bye')
print(d, e, f)
```

24.7 multiply_three_ott_comments.py

Copy the previous code, `multiply_three.py`, into a new program `multiply_three_ott_comments.py`. Write a comment above each line explaining what it does.

24.8 fahrenheit_to_celsius.py

Write a program, `fahrenheit_to_celsius.py`, that contains a function to convert Fahrenheit to Celsius.

24.9 celsius_to_fahrenheit.py

Write a program, `celsius_to_fahrenheit.py`, that contains a function to convert Celsius to Fahrenheit.

CHAPTER

TWENTYFIVE

NUMBERS [FUNC-06-PARAM-CTD.RST]

25.1 `sum_3.py`

Write and call a function **sum** that that takes 3 arguments and returns the result of adding them together.

25.2 `bigger.py`

Create a program which has a function `test_bigger(a, b)` which returns `True` if `a` is greater than `b` otherwise return `False`. This program should also have test code.

CHAPTER

TWENTYSIX

FUNCTIONS - OPTIONAL/DEFAULT PARAMETERS [FUNC-07-PARAM-DEFAULT.RST]

Functions parameters can have default values which are used if no argument is provided by the call. For this exercise, use your `numbers` directory.

26.1 `geometric_score.py`

Write, fix and test the following code, **geometric_score.py**.

```
# geometric_score.py

def increase(previous_score, ratio = 0.1):
    difference = 1.0 - previous_score
    increase_amount = difference * ratio
    score = previous_score + increase_amount
    return score

def decrease(previous_score, ratio = 0.1):
    decrease_amount = previous_score * ratio
    score = previous_score - decrease_amount
    return score

score = 0.5
while(score < 0.9):
    score = increase(score)
    print(score)
```

26.2 `geometric_score_comments.py`

By expanding the calling code, for an initial score between 0.0 and 1.0 explain how the functions work. Save this code as **geometric_score_comments.py**.

CHAPTER

TWENTYSEVEN

FUNCTIONS 3.1, SCOPE 1 AND VARIABLES - INTRODUCTION TO *SCOPE* [FUNC-08-SCOPE.RST]

Scope concerns the *visibility* of names in your code. The *scope* of a variable in a program is the lines of code in the program where the variable can be accessed. For example, a variable name with *global* scope can be seen and accessed anywhere in a file. Generally, avoid using global variables but if you must then try not to change them. A variable defined in a function is *local* to that function. Scope is better understood with practice.

27.1 scope_01.py

Predict the output of the following code snippet and test your answer.

```
x = 1 # Global (file) scope.

def scope_function():
    x = 2 # Local (function) scope.
    print(x)
print(x)
scope_function()
print(x)
```

27.2 scope_02.py

Predict the output of the following code snippet and test your answer.

```
x = 1 # Global (file) scope.

def scope_function():
    print(x)
print(x)
scope_function()
print(x)
```

CHAPTER

TWENTYEIGHT

FUNCTIONS 3.2, SCOPE 2 AND VARIABLES - `GLOBAL` [FUNC-09-SCOPE-GLOBAL.RST]

Things that you create inside a function are only accessible inside a function. The opposite is not true, you can, if the conditions are right, access things sitting outside of a function from within a function without them being passed in as an argument. However, despite being able to access, there can be complications with changing an external object. If you do want to change something outside the function you will have to use **global**. Try to avoid overusing **global**.

Global variables should be used sparingly.

28.1 `scope_03.py`

```
# scope_03.py

x = 1 # Global (file) scope.

def scope_function():
    global x
    x = 2     # Global (file) scope.
    print(x)
print(x)
scope_function()
print(x)
```

28.2 `scope_04.py`

Before running, predict what the output of this program will be. You must understand this before moving on.

```
# scope_04.py

x = 1
print(x)
def fun1():
    print('fun1')
    print(x)

def fun2():
    print('fun2')
```

(continues on next page)

(continued from previous page)

```
    x = 100
    print(x)

print(x)
fun1()
fun2()
x += 1
print(x)
fun1()
fun2()
```

28.3 scope_05.py

Write this new version of the previous code. After running, fully explain, in comments, what is going on.

```
# scope_05.py

x = 1
print(x)
def fun1():
    print('fun1')
    print(x)

def fun2():
    print('fun2')
    global x
    x = 100
    print(x)

print(x)
fun1()
fun2()
x += 1
print(x)
fun1()
fun2()
```

28.4 scope_06.py

What is the output of this code?

```
# scope_06.py

number = 0

def func():
    global number
    number += 1
```

(continues on next page)

(continued from previous page)

```
func()
func()
func()
print(number)
```

CHAPTER

TWENTYNINE

FUNCTIONS - MUTABLE OBJECT DEFAULT PARAMETER [FUNC-10-PARAM-DEFAULT-MUTABLE.RST]

There are situations where the behaviour default parameters may take you by surprise. In a program a functions default parameters are evaluated the first time a function is called. Subsequent calls use the previously evaluated value. If a parameter is mutable object, such as a list, dictionary, or instances of most classes, then the default parameter value will be that left by the previous call.

29.1 default_parameter_accumulate.py

The following function accumulates the arguments passed to it on subsequent calls:

```
# default_parameter_accumulate.py

def f(a, L=[]):

    L.append(a)

    return L



print(f(1))
print(f(2))
print(f(3))
```

This will print

```
[1]
[1, 2]
[1, 2, 3]
```

29.2 default_parameter_no_accumulate.py

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
# default_parameter_no_accumulate.py

def f(a, L=None):
    if L is None:
        L = []

    L.append(a)

    return L

print(f(1))
print(f(2))
print(f(3))
```

This is output

```
[1]
[2]
[3]
```

CHAPTER

THIRTY

FUNCTION CALLING AND HELP [HELP-01-HELP.RST]

30.1 Functions 1 - calling

A function is a technique for grouping and reusing code. It is a predefined sequence of statements which can be *called* to carry out a computation. Functions often *take* arguments and *return* results. Python comes with some **built-in** functions. To **call** a function you use its name and parentheses - curved brackets. Indeed, you have already encountered Python's `print` function.

`help()`, is another example of a **built-in** function.

30.2 Help 1 - `help` function and interactive help

Including on-line documentation, tutorials, forums and printed books, there are many ways to find Python help. If you have a Python problem, as it is likely that someone else has already had a similar issue, your **first port of call is an internet search engine**. However, there is also interactive help built into your Python session. (Note, interactive help is not always installed.) In this section you will be investigating Python's built in help function.

Note for later. For `help` to work on a *statement* the *statement* needs to be in quotes. This is an aside and detail that you can put at the back of your mind.

30.3 Interactive help

Try this in the **Python** (>>>) terminal.

```
>>> help()
help>
    ...
```

Type `q` and `return` to quit interactive `help`.

```
help>q
```

To find help on a particular topic, e.g. `print`:

```
>>> help(print)
```

Help on an type or object, e.g. `1.1`:

```
>>> help(1.1)
```

CHAPTER

THIRTYONE

TYPES 1 - ``TYPE` FUNCTION [HELP-02-TYPE.RST]

In Python, values have particular `types` and Python can tell you the **type** of a value (and other objects) with the `type` built-in-function.

31.1 `Interactive types

- Letters, spaces and other characters contained within quotes, for example 'Hello, world!', are **str** (strings of characters).

  `>>> type("python")`

  `<class 'str'>`

  `>>> type('123.0')`

  `<class 'str'>`

- In Python the numbers -1, 0, 1, 2 and the like are `int` (integers).

  `>>> type(123)`

  `<class 'int'>`

- Numbers such as 1.2, 1.0, etc. are `float` (floating point numbers).

  `>>> type(123.0)`

  `<class 'float'>`

31.2 `one_type.py`

Write this program but try to guess the output before you run the code.

```
# one_type.py

print(type(1))
print(type('1'))
print(type(1.0))
print(type(1 + 1))
```

31.3 `one_type_ott_comments.py`

Saving as `one_type_ott_comments.py` , prove your understanding by copying and commenting the previous `one_type.py` .

31.4 Types 2 - coerce into common type `coerce_type.py`

Write a program, `coerce_type.py` , that outputs the type of the following operations.

```
print(type(1 * 2))
print(type(1.0 * 2))
print(type("one" * 2))
```

Add comments to your code explaining what is going on. You might have to experiment.

CHAPTER

THIRTYTWO

HELP 2 [HELP-03-DIR.RST]

32.1 *dir*

Typically, used during an *interactive session*, `dir` is another useful way of finding help during your Python session. It will tell you what names are in your *local scope* or *attributes* for objects. Try the following.

- First, launch Python in interactive mode.

  ```
  > python
  ```

- Then, in interactive mode…

  ```
  >>> dir() # names in local scope
  ```

  ```
  >>> dir(1.1) # attributes of a float
  ```

  ```
  >>> dir("hi") # attributes of a string
  ```

  ```
  >>> help(dir) # more information on dir
  ```

CHAPTER

THIRTYTHREE

COMMENTS 2 - TRIPLE QUOTES [HELP-04-TRIPLE_QUOTES.RST]

As well as the `#`, another way of creating a comment is to surround the comment with triple quotes (`""" """` or `''' '''`). A triple quotes comment can be spread over multiple lines.

33.1 `comments_multiline.py`

Write and test a triple quote program, `comments_multiline.py`.

```
""" Triple quotes
 for multi-line comments """
```

33.2 `comments.py`

Predict then test the output of this code.

```
""" One, two,
 three, """
print ("four, five", "seven") # eight
# nine
print("10", """ eleven
12""")
```

CHAPTER THIRTYFOUR

COMMENTS 2.1 - TRIPLE QUOTES

As well as multi-line comments, triple quotes can also be used for displaying a multi-line message.

CHAPTER

THIRTYFIVE

PRINT_MULTILINE.PY

What happens if you start your first line with `print`, `print_multiline.py`?

```
print(""" Triple quotes
 for multi-line comments """)
```

CHAPTER

THIRTYSIX

HELP 3 [HELP-05-PYDOC.RST]

36.1 pydoc

While the internet and search engines are the first place you will look for Python help and information, pydoc, a module for generating documentation, is another option. In fact, the interactive help you used earlier, uses pydoc behind the scenes.

You can run 'pydoc' from your computer to view the help from outside Python's interactive mode. From your computer's command line, try this.

```
> pydoc sys
```

CHAPTER

THIRTYSEVEN

COMMENTS - *DOCSTRING* [HELP-06-DOCSTRING.RST]

A **docstring** is a *string* at the beginning of a module, function, class, or method definition. The **docstring** then becomes the special attribute `__doc__` for the particular *object*.

37.1 `hello_help.py`

Comments - functions *docstring*,

Create your own help with docstring. Write and this program, `hello_help.py`.

```
# hello_help.py

def my_func():
    """ Gives the world a
    friendly message
    """

    print("hi")

print(my_func.__doc__)
```

37.2 Test your docstring

Test the program in this manner. Note the **i**.

> `> python -i hello_help.py`
>
> `>>> my_func()`
>
> `>>> help(my_func)`

CHAPTER THIRTYEIGHT

[HELP-07-DOCSTRING.RST]

```python
def my_function():
    pass

help(my_function)
```

```
Help on function my_function in module __main__:

my_function()
```

```python
def my_function():
    """This is a function that does nothing.


    blah, blah...
    """
    pass

help(my_function)
```

```
Help on function my_function in module __main__:

my_function()
    This is a function that does nothing.


    blah, blah...
```

CHAPTER

THIRTYNINE

LOGIC OPERATOR V ASSIGNMENT OPERATOR [IF-01-RELATIONAL.RST]

39.1 Logic 2, relational operators == and !=, and assignment =

Relational operations result in either `True` or `False`.

- Gotcha, keep in mind that = is an **assignment** operator and == is a **relational** operator.
- The following exerciser uses the operator == to compare the objects on either side - `True` if they are equal.
- `!=` compares the objects on either side - `True` if they are not equal.

39.2 `relational_01.py`

Solve these logic problems, first by hand then verify your results by running the program.

```
print(1 + 1 == 2)
print(1 + 1 == 3)
print(1 + 1 != 3)
print('right' == 'left')
print('right' == 'right')

a = 1 + 1
b = 2
print(a == b)
```

39.3 `relational_02.py`

Do you understand the problem with this?

```
print(1 == 2)
print(1 = 2)
```

CHAPTER

FORTY

IF 1 - INTRODUCTION [IF-02-IF.RST]

The `if` statement is used to check a condition and selectively execute a block of code. The `if` block is indicated by indentation (recommended 4 spaces).

40.1 if_01.py

Write and test this program.

```
left_var = 15
right_var = 3 * 5

if left_var == right_var:
    print("They are the same value.")
    print("Inside the 'if' block.")

print("End of program.")
```

40.2 if_02.py

Make a copy of `if_01.py` and save as `if_02.py` .

- Change the == into a `!=`.
- Make the `print` messages appropriate.
- Change the `5` to `"5"`.
- Comment your code to demonstrate that you understand what is happening.

CHAPTER

FORTYONE

LOGIC [IF-03-LOGIC_RELATIONAL.RST]

41.1 logic_a_b.py

Given that `A = 5` and `B = 6`, predict the output before running the following:

```
A = 5
B = 6
print(not(A > B) and (A < B))
print(not(True) or (A > B))
```

41.2 logic_1.py

Try to figure out the answer to each question before you run the code.

Given that **a = 1** and **b = 2**, are the following print outputs **True** or **False**?

```
a = 1
b = 2

# 1
print(a > b)

c = a + a

# 2
print(c > b)

c = a + a

# 3
print(c <= b)

# 4
print(b != a)

# 5
c = b
d = a + a
```

(continues on next page)

(continued from previous page)

```
# 6
print(c != d)

d = a + a + a
c = a + a + b

# 7
print(d == c)

c = a - b
d = b - c

# 8
print((b + d) != (a + c))

c = a
d = c - a

# 9
print(d <= c)

c = 3
d = 2

# 10
print((a + c) < (b + d))

c = 12
d = a + b
c = c + d

# 11
print(c >= (d + 4))
```

41.3 Logic - logic and relational operators `logic_and_relations.py`

Before running, predict the output of this program, `logic_and_relations.py`.

```
print(True)
print(False and False)
print(False and True)
print(True and False)
print(True and True)
print(True or 1 == 2)
print(not (True and True))
print(2+1 == 3 and not("left" == "right" or "good" != "fun"))
print(1 + 1 == 2 and 1 + 1 != 3)
print(1 + 1 == 2 and 1 + 1 == 3)
```

CHAPTER

FORTYTWO

IF 2 - IF, ELSE [IF-04-ELSE.RST]

In this section the `if...else` statement is introduced. When the `if` condition is false the `else` block is executed.

42.1 if_else.py

Write and test this program.

```
var = 'a'
if var == 'a':
    print("We have an 'a'.")
else:
    print("We don't have an 'a'.")

print("That's the end of the 'a' test.")
```

42.2 if_else_not_a.py

Saving as `if_else_not_a.py`, copy the previous program, change the variable value from `a` to another value and test. Finally, explain your code with comments.

CHAPTER

FORTYTHREE

LOGIC RELATIONAL [IF-04-LOGIC_RELATIONAL.RST]

43.1 logic_2.py

Predict and test the output of this code.

```
# logic_2.py

print("True or False = ", True or False)
print("7 <= 7 is ", 7 <= 7)
print("True and False = ", True and False)
print("5 != 5 is ", 5 != 5)
```

43.2 logic_likes.py

Fix the code by adding either **True** or **False** after the comma in the **print_answer** calls. The correct output from the function should be **True** in each case.

```
def print_answer(logic_1, logic_2):
    print(logic_1 == logic_2)

i_like_pizza = True
i_like_milk = False
i_like_swimming = True
i_like_football = False
i_like_tea = True

print_answer(i_like_tea, True)
print_answer(i_like_swimming or i_like_milk, )
print_answer(not(i_like_pizza or i_like_milk), )
print_answer(i_like_pizza and i_like_football == True, )
print_answer(not (1 != 2 and 1 == 2), )
```

43.3 if - sheep and wolves `if_exercise.py`

Write and test this program, **if_exercise.py**. Write it iteratively, testing a few lines at a time.

```
# if_exercise.py

people = 30
sheep = 30
wolves = 25
if people < sheep:
    print("More sheep than people.")
if people > sheep:
    print("Not more people than sheep.")
if people < wolves:
    print("Fewer people than wolves.")
if people > wolves:
    print("More people than wolves.")
wolves += 5
if people >= wolves:
    print("More or equal people and wolves.")
if people <= wolves:
    print("People are fewer or equal to wolves.")
if people == wolves:
    print("The same number of people and wolves.")
```

CHAPTER

FORTYFOUR

IF 3 - IF, ELIF AND ELSE [IF-05-ELIF.RST]

Within an `if` / `else` clause you can insert multiple `elif` statements.

44.1 travel_info.py

Write and test this program, save as `control/travel_info.py`.

```
people = 30
train_seats = 40
if train_seats > people:
    print("Take the train.")
elif train_seats < people:
    print("Do not use the train.")
else:
    print("Not sure.")
```

44.2 ott_comments_travel_info.py

Write a fully (over) commented version of `travel_info.py` and save as `ott_comments_travel_info.py`.

44.3 travel_info2.py

When there are multiple code branching being controlled by `elif`s, it can be confusing. Try to keep your code simple and easy to understand. The following is not good, clear code.

Make a copy of `travel_info.py` and save as `travel_info2.py`.

- Add another `elif` statement which advises to take the train if you are running late.
- Change the number of people.
- Add a boolean for running late.
- Explain what is going on.

```
people = 40
train_seats = 40
running_late = True
if train_seats > people:
```

(continues on next page)

(continued from previous page)

```
    print("Take the train.")
elif train_seats < people:
    print("Do not use the train.")
elif running_late:
    print("Just take the train. You are late.")
else:
    print("Not sure.")
```

CHAPTER

FORTYFIVE

IF REVISION [IF-06-REVISION.RST]

45.1 zombie_escape.py

Project - More *if* practice - zombie escape game. Write, test and comment the following program, save as **zombie_escape.py**.

```
print("Terrible moaning at your front door.")
print("There are two windows.")
print("Do you get out through window #1")
print("or window #2 to the back?")
window = input("> ")

if window == "1":
    print("There's a large Zombie here.")
    print("What do you do?")
    print("1. Try to cure the ailment.")
    print("2. Shout at the Zombie.")
    large = input("> ")

    if large == "1":
        print("The Zombie eats your face off.")
        print("Just brilliant!")
    elif large == "2":
        print("The Zombie eats your legs off.")
        print("Just brilliant!")
    else:
        print("Well, doing %s is probably better." % large)
        print("Zombie stumbles away.")
elif window == "2":
    print("Here's a Zombie horde.")
    print("1. Spin.")
    print("2. Jump.")
    print("3. Hope.")
    horde = input("> ")

    if horde== "1" or horde == "2":
        print("You survive by...")
        print("covering yourself in chocolate.")
        print("Just brilliant!")
    else:
```

(continues on next page)

(continued from previous page)

```
        print("You become a Zombie and rot.")
        print("Just brilliant!")
else:
    print("You freak and die of heart failure.")
    print("Just brilliant!")
```

CHAPTER

FORTYSIX

INDEXING 1 [INDEX-01.RST]

46.1 Strings 4 - indexing

Characters in strings are accessed using square brackets. The first element is at the index zero, e.g. `a_string[0]`.

46.2 `alphabet_hello.py`

Write and test the following code, `alphabet_hello.py`.

```
a_str = "abcdefghijklmnopqrstuvwxyz"
print(a_str[7] + a_str[4] + a_str[11] + a_str[11] + a_str[14])
```

46.3 `alphabet_my_name.py`

Write a program `alphabet_my_name.py`. This should use the same `a_str` and indexing technique as `alphabet_hello.py` to display your name.

46.4 Lists 1 - lists introduction and indexing

A Python **list** is a container for things (a grouping of objects) in an organised order. A list is created using square, brackets i.e. `[]`. You will use them often.

46.5 `empty.py`

Create an empty list.

```
an_empty_list = []
print(an_empty_list)
```

46.6 ten.py

Create and access a list of 10 integers 0 to 9. Write and test `ten.py`.

```
ten_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(ten_list)
print(ten_list[0])
print(ten_list[6])
```

46.7 ten_ott_comms.py

Explain `ten.py` with comments in `ten_ott_comms.py`.

46.8 Lists 2 - mutable

You can access and change items in a list. (Note, you can not change letters of a string in place.)

46.9 list_three.py

Create and run this program.

```
a_list = ["one", "two", "three"]

print(a_list)
print(a_list[1])
print(a_list[1][0])

a_list[1] = 2
print(a_list)
```

46.10 list_three_ott_comms.py

Explain `list_three.py` with comments in `list_three_ott_comms.py`.

46.11 Tuples 1

A `tuple` is another Python container. Like `lists` you access members by position, however, you can not modify the members of a **tuple**. A **tuple** is created by separating data with commas separating entries. A **tuple** is often delimited by brackets i.e. `()`.

46.12 tuple_simple.py

Try this program.

```
t1 = (1, 2, 3)
t2 = (4, 5, 6, 7)
t3 = t1 + t2
print(t1, t2, t3)
```

46.13 Dictionaries 1

A Python **dictionary**, similar to a **list**, is a container for objects. Unlike lists, which you access by position, dictionary members are accessed by **key** values. A dictionary is delimited with curly brackets i.e. **{}**.

46.14 french_01.py

Write, test and comment the following program, `french_01.py`.

```
french = {'yes':'oui', 'no':'non', 'hello':'bonjour'}
print(french['hello'])
```

46.15 french_02.py

Write a program `french_02.py` that uses the dictionary

`french = {'yes':'oui', 'no':'non', 'hello':'bonjour'}`

to display `ouiouioui`.

46.16 french_03.py

Write, test and explain in comments this sweet program,`french_03.py`.

```
french = {'yes':'oui', 'no':'non', 'hello':'bonjour'}
print(french['hello'][3]
      + french['yes'][1]
      + french['no'][2]
      + french['yes'][2]
      + french['no'][1]
      + french['hello'][-1])
```

CHAPTER

FORTYSEVEN

INDEXING 2 [INDEX-02-CTD.RST]

Try to figure out the answer to each question before you run the code. You do not have to do these questions from memory. Reference your notes, Internet and use the Python interactive prompt. There will be some new material. To understand the code you may need to carry out some of your own investigation, including on-line.

47.1 list_123.py

Create a list of strings `one`, `two` and `three`.

47.2 lists_0_to_9.py

Create a list of 10 integers 0 to 9.

47.3 cakes_animals.py

Using this **dictionary** elements and only two accessors and one **operator** create the output `tttttt`.

```
dict_1 = {'cakes':['jaffa', 'sponge'],
    'animals':['cats', 'dogs'],
    'numbers':[1, 3, 6, 9]}
```

CHAPTER

FORTYEIGHT

INDEXING 3 [INDEX-03-CTD.RST]

48.1 lists.py

What is the output of this code snippet? Try to figure out the answer before you run the code.

```
# list_indexlist_index.py

six_list = [1,2,3,4,5,6]
print(six_list[0:4])
```

48.2 list_operators.py

List operations. Before verifying, predict the output of the following:

```
# list_operators.py

name_list = ['John', 'Jane', 'Jean'] + ['Graeme', 'George']
print(name_list)
print(name_list * 2)
```

48.3 alphabet_negative.py

Strings - negative indexing

Figure out what is going on and comment the code, `alphabet_negative.py`.

```
# alphabet_negative.py

alphabet_string = "abcdefghijklmnopqrstuvwxyz"
print(alphabet_string[0])
print(alphabet_string[-1])
print(alphabet_string[25])
print(alphabet_string[13])
print(alphabet_string[-13])
```

48.4 emergency.py

Lists - negative indexing

Write, test and explain with comments this program, `emergency.py`.

```
# emergency.py

numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(numbers[-1], numbers[-1], numbers[-1])
```

48.5 dict_somewhat_complex.py

What is the result of this somewhat complex **dictionary** accessors.

```
d = {'a':1,'b':2,'c':3,'d':{'one':[1,'two']}}

print("d['d']['one'][-1][2] = ",d['d']['one'][-1][2])
```

CHAPTER

FORTYNINE

INDEXING 4 - EXERCISES [INDEX-04-CTD.RST]

49.1 len.py

Sequences - `len` function

The built-in `len` function returns the numbers of items of sequence.

For example:

```
# len.py

print(len([1, 2, 3]))
print(len("Length"))
```

Gives:

```
3
6
```

49.2 complex_len.py

Using only the **output** of the `len` built-in function and the list (`complex_list`)

```
complex_list = ["Hello", {'a': [1, 2, 3, 4], 'b':(1,), 'c': []}]
```

write a program, `complex_len.py`, that gives the output.

```
0
1
2
3
4
5
```

49.3 mixed_list.py

Create a mixed list, a list of numbers, strings and even another list.

```
# mixed_list.py

list_various = [1, 2, 3, 'abc', [1, 'a']]
print(list_various)
print(list_various[4][0])
```

49.4 mixed_list_ott_comms.py

Explain **mixed_list.py** with comments in **mixed_list_ott_comms.py**.

49.5 mixed_dic.py

Run this program, `mixed_dic.py`.

```
# mixed_dic.py

dic = {'a':1,'b':'two','lst':[10,9,8]}
print(dic)
print(dic['a'])
print(dic['b'])
print(dic['b'][1])
print(dic['lst'][len(dic['lst'])-1])
```

49.6 mixed_dic_ott_comments.py

Make a copy of `mixed_dic.py` and save as `mixed_dic_ott_comments.py`. Write a comment explaining each line of the output. The last line will be easier to understand if you break-up into several lines and experiment on the Python interpreter with the *built-in* **len()** function.

49.7 complex_dic.py

Dictionary

Using the following dictionary `dic_1` as the data source, write a program `complex_dic.py` that gives the output below. (Hint: `int()`)

```
dic_1 = {'a':'a', 'b':2.2, '3':'a str', 'list':['one', 100, 3.3]}
```

> > python complex_dic.py
>
> 2.2

2

t

n

CHAPTER

FIFTY

INDEXING - SLICING [INDEX-05-CTD.RST]

50.1 alphabet_slicing.py

To create sub-strings you can use string *slicing*. Comment and explain this code, `alphabet_slicing.py`.

```
alphabet_string = "abcdefghijklmnopqrstuvwxyz"
print(alphabet_string[0:5])
print(alphabet_string[:5])
print(alphabet_string[1:5])
print(alphabet_string[-25:5])
print(alphabet_string[-2:])
print(alphabet_string[24:])
print(alphabet_string[24:26])
print(alphabet_string[0:13:2])
```

50.2 alphabet_slicing_2.py

Using slicing of `alphabet_string = "abcdefghijklmnopqrstuvwxyz"` give the following outputs:

```
bcdefghijklmnopqrstuvwxyz
adgjmpsvy
ghijklmnopqr
```

50.3 glasgow.py

Using the string `Glasgow` and only two string slices, create the new string `gowGla`.

50.4 Lists 4 - slicing `list_slicing.py`

Write, test and comment this program, `list_slicing.py`, which is an example of lists and *slices*. Try to predict the output before you run your code.

```
number_list = [1, 2, 3, 4, 5]
print(number_list[1:3])
print(number_list[0:3])
print(number_list[0:5:2])
print(number_list[1:])
print(number_list[-2:5])
```

50.5 Lists 4 - slicing `two_three_four_slice.py`

Write your own slicing program, using `number_list = [1, 2, 3, 4, 5]` and the number `-1` and `1` in a way to output `[2, 3, 4]`. Save your program as `two_three_four_slice.py`.

CHAPTER

FIFTYONE

INDEXING - LIST METHODS [INDEX-06-CTD.RST]

An object, such as a list, can have internal functions called methods. They are accessed using *dot* notation.

51.1 list_methods.py

Write and test this program, `list_methods.py`.

```
list_of_numbers = [1, 2, 3, 4]
list_of_numbers.append(1)
list_of_numbers.pop()
list_of_numbers.extend([11, 12, 13])
list_of_numbers.pop(2)
list_of_numbers.reverse()
list_of_numbers.insert(3, 21)
list_of_numbers.sort()
print(list_of_numbers)
```

51.2 list_methods_ott_comments.py

Make another copy of `list_methods.py`, comment each line and save as `list_methods_ott_comments.py`. Use `print` to understand what each line of this program does.

51.3 list_complex.py

Nested lists and append method. Write and test `list_complex.py`. You may be surprised to find that this program outputs `Glasgow is hot!`.

```
nested_list = []
nested_list.append('glasgow')
nested_list = nested_list + [ 1 , 1.0, [3, 5], 7]
nested_list.append(['is', 'wet'])
nested_list[5][1] = 'hot'
print(nested_list[0].capitalize(), nested_list[5][0], nested_list[5][1] + "!")
```

51.4 list_complex_ott_cmts.py

Copy `list_complex.py`, carefully comment and save as `list_complex_ott_cmts.py`.

51.5 list_complex2.py

Write a program, `list_complex2.py`, that uses `nested_list = [1, [200, 300], 4]` to write `1 200`.

51.6 list_range.py

List and `range`. Predict and test the result of this code?

```
numbers = range(1, 10, 2)
print("numbers[1:] = ", list(numbers[1:]))
```

51.7 lists_7.py

Lists `len` and modulo. Given the two lists, below, which of the following 4 code snippets give the result `7`? Try to figure out the answer before testing.

```
list_1 = [0, 1, 2, 3, 4, 5, 6, 7]
list_2 = [1, 2, 3, 4, 5, 6, 7]

print(len(list_1))
print(len(list_2))
print(len(list_1) % len(list_2))
print(len(list_2) % len(list_1))
```

51.8 out_of_sorts.py

Lists and sort. What do think the output of this code will be? Try to gess the answer before running the code.

```
out_of_sorts = [1,3,2,4,3,5,4,6,5,7]
out_of_sorts.sort()
print("out_of_sorts.sort()", out_of_sorts)
```

51.9 `list_reverse_method.py`

List reverse method. Write a program that uses a list method to change `[1, 2, 3, 4, 5, 6]` to `[6, 5, 4, 3, 2, 1]`. Display the start list and result.

CHAPTER

FIFTYTWO

INPUT 1 - INTRODUCTION [IO-01-INPUT.RST]

52.1 input_name.py

Write and test the following program, input_info.py.

```
print("What is your first name?")
name = input()

print("Hello ", name)
```

52.2 input_age.py

Write a similar program to input_name.py that asks for and reads in a user's age, input_age.py .

CHAPTER

FIFTYTHREE

INPUT - INPUTTING DATA AND % STRING FORMATTING EXPRESSION [IO-02-INPUT.RST]

The % operator is used to created formatted strings.

53.1 input_info_1.py

Write and test the following program, input_info_1.py.

```
name = input("What is your first name? ")
surname = input("What is your surname? ")
age = input("How old are you? ")

print("Hello %s %s, you are %s years old." % (name, surname, age))
```

53.2 input_info_2.py

Write a program input_info_2.py to ask a user a question.

CHAPTER

FIFTYFOUR

INPUT [IO-03-INPUT.RST]

54.1 Revision - `data input: pause

How can you pause a program until you hit the return key?

1. `print(pause)`
2. `input()`
3. `wait()`
4. `pause()`

```
def circular():
    """This loops over abc forever."""
    while True:
        for connection in ['a', 'b', 'c']:
            yield connection
```

```
c = circular()
```

```
next(c)
```

```
'c'
```

```
c.__next__()
```

```
'b'
```

```
def my_next(i):

    return ['a', 'b', 'c'][i]
```

```
circle over 0 1 2

use %
```

CHAPTER FIFTYFIVE

LINKS

Moving links out of the body of the text. I don't want to have to check the links work and are up to date.

For more on the interactive prompt see:

- http://openbookproject.net/thinkcs/python/english3e/way_of_the_program.html

For more information on `help`

- http://openbookproject.net/thinkcs/python/english3e/iteration.html#help-and-meta-notation

For more information on running a program see:

- http://openbookproject.net/thinkcs/python/english3e/way_of_the_program.html

For more information on errors see:

- http://openbookproject.net/thinkcs/python/english3e/way_of_the_program.html#syntax-errors

For more information on comments see:

- http://openbookproject.net/thinkcs/python/english3e/way_of_the_program.html#comments

For some more information on arithmetic operators

- http://openbookproject.net/thinkcs/python/english3e/variables_expressions_statements.html

For more about the modulus operator see:

- https://en.wikipedia.org/wiki/Modulo_operation
- http://openbookproject.net/thinkcs/python/english3e/variables_expressions_statements.html#the-modulus-operator

More on conditionals:

- http://openbookproject.net/thinkcs/python/english3e/conditionals.html
- http://openbookproject.net/thinkcs/python/english2e/ch04.html

More on conditional execution:

- http://openbookproject.net/thinkcs/python/english3e/conditionals.html#conditional-execution
- http://openbookproject.net/thinkcs/python/english3e/conditionals.html

For more information on string operations see

- http://openbookproject.net/thinkcs/python/english3e/strings.html#strings

Function

- http://openbookproject.net/thinkcs/python/english3e/fruitful_functions.html
- http://openbookproject.net/thinkcs/python/english2e/ch03.html#parameters-arguments-and-the-import-statement

- http://openbookproject.net/thinkcs/python/english3e/functions.html

For more on lists:

- http://openbookproject.net/thinkcs/python/english3e/lists.html

For more on files see:

- http://openbookproject.net/thinkcs/python/english3e/files.html?highlight=readline#reading-a-file-line-at-a-time

More on exception handling can be found here:

- http://openbookproject.net/thinkcs/python/english3e/exceptions.html#catching-exceptions

Class

- http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

For more on `for` loops

- http://www.greenteapress.com/thinkpython/html/thinkpython009.html#toc88

For more on data types:

- http://openbookproject.net/thinkcs/python/english3e/variables_expressions_statements.html

For more on files:

- http://openbookproject.net/thinkcs/python/english3e/files.html

For more on keyboard input:

- http://www.openbookproject.net/thinkcs/python/english2e/ch04.html#keyboard-input

More on classes and parameters:

- http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

For more on exceptions:

- http://openbookproject.net/thinkcs/python/english3e/exceptions.html#raising-our-own-exceptions

For more on `pydoc` see:

- https://docs.python.org/3/library/pydoc.html
- See https://www.python.org/dev/peps/pep-0257/ for conventions used for **docstring**s.

http://openbookproject.net/thinkcs/python/english3e/strings.html#the-string-format-methodm

For more of `while` see:

- http://openbookproject.net/thinkcs/python/english3e/iteration.html#the-while-statement

For more on `for`:

- http://openbookproject.net/thinkcs/python/english3e/iteration.html#the-for-loop-revisited

For more on continue see:

- http://openbookproject.net/thinkcs/python/english3e/iteration.html#the-continue-statement

For more in modules see:

- http://openbookproject.net/thinkcs/python/english3e/modules.html#modules

For more on `import`:

- http://openbookproject.net/thinkcs/python/english3e/modules.html#term-import-statement

For more information see:

- https://docs.python.org/3/library/operator.htmlS

For more information on Tkinter see http://docs.python.org/2/library/tkinter.html.

- http://docs.python.org/2/library/time.html
- http://docs.python.org/2/library/datetime.html
- http://docs.python.org/2/library/calendar.html

For more on using the `len` function:

- http://openbookproject.net/thinkcs/python/english3e/strings.html#length
- http://openbookproject.net/thinkcs/python/english3e/lists.html#list-length

CHAPTER

FIFTYSIX

LISTS AND LIST COMPREHENSION [LISTS-01-COMP.RST]

```
a_list = []

for i in range(10):
    a_list.append(2*i)
```

```
a_list
```

```
b_list = [ 2*i for i in range(10) ]
```

```
b_list
```

CHAPTER

FIFTYSEVEN

LOOPING 1 - A SIMPLE WHILE LOOP [LOOP-01-WHILE.RST]

A `while` loop will keep executing a code *block* of code as long as the boolean expression controlling it is `True`. Although useful, `while` loops are not used in Python as much as other languages.

57.1 simple_while_loop.py

Write, test and comment each line of this code, `simple_while_loop.py`.

```
x = 0
while x < 10:
    x = x + 1
    print(x)
```

57.2 while_two_step.py

Copy the previous `simple_while_loop.py` program and save as `while_two_step.py`. Make modifications so that the output is:

```
2
4
6
8
10
```

CHAPTER

FIFTYEIGHT

LOOPING 2 - FOR LOOP [LOOP-02-FOR.RST]

In Python, a `for` loop, loops over the items in any sequence. The `for ... in ... :` statement manages the process and stops when there are no items left. `for` loops are extremely useful in Python and you will use them often.

58.1 `simple_for.py`

Write and test the the following program, `simple_for.py`.

```
list_of_names = ["Jim", "Julia", "John"]

for name in list_of_names:
    print("Hello", name)
```

58.2 `int_for.py`

Fix, test and comment the following `int_for.py` program. Hint: `[1, 2, 3]`.

```
for item
    print(item + 1)
```

The output should be:

```
2
3
4
```

CHAPTER

FIFTYNINE

LOOPS [LOOP-03.RST]

59.1 while_3.py

What do you think the output of this code fragment will be? Test your answer.

```
x = 0
while x < 3:
    print(x)
    x += 1
```

59.2 while_123.py

Use a `while` loop over `three_list = ["one", "two", "three"]` to display each string to the console.

59.3 types_in_list.py

Write a short program that uses a `for` loop to display the type of each member of the list `[123.4, "one two three four", 1234]`.

CHAPTER SIXTY

LOOPING 2 [LOOP-04.RST]

In this exercise you are going to write two programs. They will both achieve the same outcome but one uses a `while` loop whereas the other uses a `for` loop.

The out put of both programs should be:

> PPyytthhoonn

60.1 expand_string_while.py

First, write test and comment this program.

```
string_in = "Python"
string_out = ""

character_position = 0
while character_position < len(string_in):
    char = string_in[character_position]
    string_out = string_out + 2 * char
    character_position = character_position + 1

print(string_out)
```

60.2 expand_string_for.py

Now, write, test, comment and compare this version of the program. Every time through the `for` loop, the next character in the string is assigned to the variable char. The `for ... in ... :` statement manages the process and stops when there are no characters left.

```
string_out = ""
for char in string_in:
   string_out = string_out + 2 * char

print(string_out)
```

CHAPTER

SIXTYONE

LOOPS - CONTINUE [LOOP-05.RST]

The continue statement allows the program flow to jump to the top of the containing loop.

61.1 continue_loops.py

Write and test this program, continue_loops.py. Add some judicious comments to explain what is going on.

```
alphabet_string = "abcdefghijklmnopqrstuvwxyz"

index = 0
while(index < len(alphabet_string) - 1):

    index = index + 1
    if (index % 2 == 0):
        continue
    print(alphabet_string[index])


count = 0
for letter in alphabet_string:

    count = count + 1
    if(count % 2 == 0):
        continue
    print(letter)
```

CHAPTER SIXTYTWO

LOOPING REVISION [LOOP-06.RST]

Try to figure out the answer to each question before you run the code.

62.1 list_method_join.py

What it the output of this piece of code?

```
# list_method_join.py

five_list = [1,2,3,4,5]
six_nine_list = [6,7,8,9]
joined_list = five_list + six_nine_list
print("joined_list = ", joined_list)
```

62.2 while_not_q.py

Write a program that uses a **while** loop to continually take command line input from a user. The program will exit when a **q** is input or print a suitable message.

CHAPTER

SIXTYTHREE

FUNCTIONS - WRAPPING A WHILE LOOP INTO A FUNCTION [LOOP-07.RST]

63.1 simple_while_loop.py

If you have not written this code already, write and test:-

```
# simple_while_loop.py

x = 0
while x < 10:
    x = x + 1
    print(x)
```

63.2 while_loop_function.py

Copy and convert `simple_while_loop.py`, into a function that you can call with a variable that replaces the **10**. Save your program as `while_loop_function.py`. Call your new function from within your program with different numbers.

63.3 step_size.py

By adding another variable to the function *parameter*, that lets you change the step size (currently + 1), change your function so it increments by different amounts. Save this as `step_size.py` and test this function to see what effect your changes have made.

CHAPTER

SIXTYFOUR

LOOPING 3 - LISTS AND A SIMPLE *FOR* LOOP [LOOP-08.RST]

In Python, `for` loops are commonly used in conjunction with lists.

64.1 `simple_for_loop.py`

Write and test the following program, `simple_for_loop.py`.

```
# simple_for_loop.py

nums_1 = [-459.0, 32.0, 212.0, 10000.0]
for n in nums_1:
    print(n)
```

64.2 `ott_comments_for_loop.py`

Copy the previous code listing, **control/simple_for_loop.py**, thoroughly comment and save as **ott_comments_for_loop.py**.

64.3 `while_lt_100.py`

Write a while loop that increase a number by `2.5` times each time round to produce the output:

```
1.0
2.5
6.25
15.625
39.0625
97.65625
```

64.4 class_list_loop.py

Looping - looping over a list.

Complete the program **class_list_loop.py**. Make a full list of your class mates, loop over this list and print a friendly message.

```
# class_list_loop.py

python_class_list = ['Alastair']
for class_mate in python_class_list:
    print(class_mate)
```

CHAPTER

SIXTYFIVE

LOOPING - LISTS AND RANGE [LOOP-09.RST]

The built-in function `range` returns a `list` of integers.

65.1 ranging.py

Use the built-in function `range` to create a selection of lists of integers.

```
# ranging.py

print(range(10))
print(range(2, 20))
print(range(0, 60, 10))
print(range(0, -12, -1))
```

65.2 for_range_loop.py

You can use `range` to quickly construct a list for a for-loop to loop over. Try this:

```
# for_range_loop.py

for num in range(20):
    print(num)
```

65.3 for_range_loop_two.py

Use your previous experience of `range` to create a list `[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]` and use a for loop to display each member individually. Save your program as `for_range_loop_two.py`.

65.4 alphabet_len.py

Looping - strings…

Write this program that tells you how long the alphabet is.

```
# alphabet_len.py

alphabet_string = "abcdefghijklmnopqrstuvwxyz"
print(len(alphabet_string))
```

65.5 alphabet_len_for.py

Using a `for` loop and not `len`, write a program (`alphabet_len_for.py`) to count the length of the alphabet. For this use your `control` directory.

65.6 alphabet_even.py

Using a `for` loop, write a program, `alphabet_even.py` , to print every second letter of the alphabet.

CHAPTER

SIXTYSIX

LOOPING - BREAK [LOOP-10.RST]

The keyword `break` is used to immediately exit the containing loop.

66.1 break_loops.py

Write and test the following `break_loops.py`.

```
# break_loops.py

print("Loop 1")
num = 0
while(True):
    print(num)
    num = num + 1
    if num == 6:
        break

print("\nLoop 2")
for num in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    if num == 6:
        break
    print(num)
```

66.2 break_while.py

On the input of `q`, use a `break` statement to exit the following programs `while` loop.

```
# break_while.py

while(True):
    user_in = input()
    if user_in == 'q':
        break
```

66.3 range_10.py

List and `range`…

What is the output of this code?

```
# range_10.py

range_10 = range(10)
print("range_10 = range(10), range_10[3:] = ", list(range_10[3:]))
```

66.4 range_3_10.py

`for` and `range`…

Before running, predict the output of the following code?

```
# range_3_10.py

for i in range(3, 10):
    print(i)
```

66.5 odd_even.py

`for`, lists and modulus

Write a short program that uses a **for** loop, **range(100)** and the modulus operator (%) to display the numbers between 0 and 99 with a short message indicating if each is **even** or **odd**.

66.6 zipping.py

Lists - *`for``, ``list`` and ``zip``,*

Investigate the use of *zip* to figure out what the following code will do, `zipping.py`?

```
nine_list = [1,2,3,4,5,6,7,8,9]
nineteen_list = [11,12,13,14,15,16,17,18,19]
print("zip(nine_list, nineteen_list) = ", zip(nine_list, nineteen_list))

answer = ""
for i, j in zip(nine_list, nineteen_list):
    answer = answer + str(i) + " " + str(j) + " "
print(answer)
```

CHAPTER

SIXTYSEVEN

TODO [LOOP-11.RST]

- While loops
- Make two versions. The first without comments the second with.
- `simple_while_loop.py`
- Add a link to finding more about while loops
- `Looping-1-a-simple-while-loop`

67.1 abcd.py

`if`, `elif`, `else` and string

Use a for loop to iterate over the string `abcdefghijklmnopqrstuvwxyz` and display the following messages appropriately:

> This is an 'a'.
>
> This is an 'b'.
>
> This is an 'c'.
>
> This is an 'd'.
>
> This is not 'a', 'b', 'c', or 'd'.

67.2 sun_scot.py

`for` and `Sunny Scotland`

Create a `for` loop which iterates over a string, `Sunny Scotland`, and displays each letter.

67.3 nested.py

`for` and nested *list*.

What is the output of this **for** loop? Write your answer before testing?

```
# nested.py

l = [[1,2],[3,4],[5,6],[7,8],[9,10]]
print("l =",l)
for i,j in l:
    print(i, j)
```

67.4 hexadecimal.py

Lists `for`, **list**, `range` and `hex` conversion...

Investigate the conversion **hex** to predict the output of this code.

```
# hexadecimal.py

l=[]
for n in range(18):
    l.append(hex(n))

print("hexadecimal 0->18", l)
```

67.5 check_6.py

Write a short program that `for` loops over the list `[1, 7, 3, 3, 75, 751, 7, 8704, 7, 8]` and displays a message only if the number in the list is greater than `6`.

67.6 two_times.py

Using the list `mixed_list = [[1,2,3] , 'Glasgow', ['1', '2', '3']]` and a nested for loop (two for loops, one inside the other) give the output:

```
2 4 6 GG ll aa ss gg oo ww 11 22 33
```

67.7 alphabet_odd.py

Looping,..

Starting with a, write a program to print every second letter of the alphabet, strings_odd.py.

CHAPTER

SIXTYEIGHT

MODULES 1 - IMPORT [MODULE-01-IMPORT.RST]

In Python, a **module** is a file that contains code that can be used in your program. For example, the `math` module provides some useful mathematical functions. To bring this into your program use the `import` Python keyword.

68.1 `import math`

Try this in your Python interactive shell (>>>).

> `> python`
>
> `>>> import math`
>
> `>>> help(math)`
>
> `This module is always available. It provides access to the...`
>
> `>>> math.pi`

68.2 `trig.py`

Write and test the program `trig.py`.

```
import math
print(math.sin(math.pi/2))
```

68.3 `trig_comments.py`

Copy `trig.py`, explain what is going on with code comments and save as `trig_comments.py`.

68.4 `trig_var.py`

Using a variable for `math.pi/2`, rewrite `math.sin(math.pi/2)` over two lines, printing our your results and save as `trig_lines.py`.

CHAPTER

SIXTYNINE

MODULES 2 - IMPORT WITH FROM [MODULE-02.RST]

Below we experiment with different ways of using `import`s to bring outside code into your program.

69.1 mixed.py

This program, `mixed.py`, uses *inbuilt* (no `import`) functions, the `math` module (`import math`) and the `random` module (`import random`). Write, test and comment each line. You may need help from the Internet.

```
import math
import random

print(math.e)
print(abs(-1.1))
print(math.floor(1.5))
print(math.ceil(1.5))
print(round(1.23456))
print(round(1.23456, 3))
print(sum([1, 2, 3]))
print(math.trunc(12345.67))
print(pow(3,2))
print(3**2)
print(3.0**2)
print(math.pow(3,2))
print(math.sqrt(4))
print(divmod(7, 2))
print(random.randint(0, 9))
print(random.randint(0, 9))
```

69.2 from

`import` using `from`...

Write and test these two programs, `import_math.py` and `import_pi.py`, alternative methods for importing code.

69.3 import_math.py

```
import math
print(math.pi)
```

69.4 import_pi.py

```
from math import pi
print(pi)
```

CHAPTER

SEVENTY

MODULES [MODULE-03.RST]

More importing.

70.1 single_variable.py

Modules and your own code import. Write your own module module single_variable.py.

```
variable = 100
```

70.2 main.py

Write the program main.py that utilises your single_variable.py.

```
import single_variable

print(single_variable.variable)
```

70.3 import variable

Given a python script with only one line, variable = 1, which of the following gives an output on the interactive prompt of 1?

- A

```
import script_2
print(script_2.variable)
```

- B

```
import script_2
print(variable)
```

CHAPTER

SEVENTYONE

MODULES - IF __NAME__ == "__MAIN__": [MODULE-04.RST]

Often the start point of a program is contained in a `if __name__ == "__main__":` block at the bottom of the a Python source file. This block is only ever used in the top level source file. In all the other files below this level anything in this block is ignored. Thus this is a good place to put code that can be used for testing or code that you do not want to be imported if a top level source file were used as a sub-module.

71.1 two_functions_1.py

Write and run this program, `two_functions_1.py`.

```
# two_functions_1.py

def fun1():
    return 1

def fun2():
    return 2

a = fun1()
b = fun2()
c = fun1()
print(a + b + c)
```

71.2 two_functions_2.py

Write run this program, `two_functions_2.py`.

```
# two_functions_2.py

def fun1():
    return 1

def fun2():
    return 2

if __name__ == "__main__":
    a = fun1()
    b = fun2()
```

(continues on next page)

(continued from previous page)

```
    c = fun1()
    print(a + b + c)
```

71.3 Importing

Explain in comments what is going with these two programs.

71.4 main_1.py

Write and test this program, main_1.py.

```
# main_1.py

from two_functions_1 import *

print(fun1())
```

71.5 main_2.py

Write and test this program, main_2.py.

```
# main_2.py

from two_functions_2 import *

print(fun1())
```

CHAPTER

SEVENTYTWO

MODULES - OS MODULE [MODULE-05.RST]

The os module provides functions for interacting with your operating system.

72.1 simple_system.py

Use help and dir to investigate os interactively then try this short example and fully comment.

```
#simple_system.py

from os import getcwd, chdir

cwd = getcwd() # Return the current working directory
print(cwd)

chdir('..') # Change directory
cwd = getcwd()
print(cwd)
```

CHAPTER SEVENTYTHREE

MODULES AND I/O - READING COMMAND LINE DATA [MODULE-06.RST]

Another way to get data into your program is from the command line. The **sys** module provides **argv** *argument vector* a list of strings from the command line when you start your program.

73.1 `two_arguments.py`

Write the following `two_arguments.py` program.

```
#two_arguments.py

from sys import argv

argument_one, argument_two, argument_three = argv

print(argv)
print("The first argument, the program name:", argument_one)
print("The second variable argument:", argument_two)
print("The third variable argument:", argument_three)
```

73.2 Run the program

Run the program with two *command line arguments*.

> > python two_arguments.py 1 two

73.3 `argv_1.py`

What is the output from this code?

```
from sys import argv
print "argv[0] =", argv[0]
```

CHAPTER

SEVENTYFOUR

MODULES - FILES AND ARGV TEXT EDITOR [MODULE-07.RST]

74.1 three_line_editor.py

Create and test this simple 3 line editor program three_line_editor.py.

```
# three_line_editor.py

from sys import argv

program_name, input_filename = argv

print("Before writing to %r, its existing content will be deleted." % input_filename)
print("Exit, hit CTRL-C (^C).")
print("Continue, hit RETURN.")

input()

text_file = open(input_filename, 'w')

print("Input three lines of text each finished by RETURN.")

input_line_1 = input("1: ")
input_line_2 = input("2: ")
input_line_3 = input("3: ")

text_file.write(input_line_1)
text_file.write("\n")
text_file.write(input_line_2)
text_file.write("\n")
text_file.write(input_line_3)
text_file.write("\n")

text_file.close()
```

74.2 `three_line_editor_cmts.py`

Try to figure out what is happening in the previous code, `three_line_editor.py`. Saving as `io/three_line_editor_cmts.py`, use what you have learnt to write thoroughly comment version of the code.

CHAPTER

SEVENTYFIVE

OPERATORS - OPERATOR MODULE [MODULE-08.RST]

There are times when you would like to pass an *operator* (such as addition, +, subtraction ,-, etc.) into a function. To achieve this you can use the **operator** *module*. The **operator** module defines *functions* that correspond to *built-in* operations for arithmetic and comparison, as well as sequence and dictionary operations.

75.1 `operator_module.py

Write, test and comment this program, **operator_module.py**.

```
import operator as op

x = 2.0
y = 15

for optr_func in (op.lt, op.le, op.eq, op.ne, op.ge, op.gt):
    print('%s(%s, %s) is ' % (optr_func.__name__, x, y) , optr_func(x, y))
```

CHAPTER

SEVENTYSIX

MODULES 4 - TKINTER, __INIT__ AND __MAIN__ [MODULE-09.RST]

76.1 Modules - import *

In this exercise we will be importing using a * shorthand. This shorthand can cause problems and should generally be avoided.

76.2 callback_gui.py

Write and test the following program. Add comments, almost to every line, explaining your code.

```
from tkinter import *

class GUI_App:
    def __init__(self, root_window):
        frame = Frame(root_window)
        frame.pack()
        self.button1 = Button(frame
            , text="QUIT", fg="red", command=frame.quit)
        self.button1.pack(side=LEFT)
        self.button2 = Button(frame
            , text="Hello", command=self.hello_callback)
        self.button2.pack(side=LEFT)
    def hello_callback(self):
        print "Hello for the GUI world."
if __name__ == '__main__':
    root = Tk()
    app = GUI_App(root)
    root.mainloop()
```

76.3 Tkinter and module imports

Which of these could form the basis of a GUI?

1. from tkinter import *
2. import tkinter
3. import tkinter as gui
4. from tkinter import Tk, Frame

CHAPTER

SEVENTYSEVEN

OPERATORS - BASIC OPERATORS, NUMBERS AND MATHS [NUMBERS-01.RST]

77.1 counting_sheep.py

Write the following `counting_sheep.py` program, designed to help you sleep, then run the program in your terminal.

```
print("Number of sheep:")
print("Ewes", 30 + 30 / 6)
print("Rams", 90 - 25 * 3)
print("Count the lambs:")
print(4 + 3 + 2 - 6 + 5 * 3 - 2 / 5 + 7)

print("Is 4 + 3 less than 6 - 8?")
print(4 + 3 <  6 - 8)
print("Add 4 and 3?", 4 + 3)
print("Subtract 6 from 8?", 6 - 8)
print("That explains why it is False.")

print("Some more.")
print("Greater?", 6 > -3)
print("Greater or equal?", 6 >= -3)
print("Less or equal?", 6 <= -3)
```

77.2 counting_sheep_ott_comments.py

Make sure you understand what is going on with the previous program. The order in which the operators are applied can impact on the answer.

- Make a copy of `counting_sheep.py` and save as `counting_sheep_ott_comments.py`.
- Above each line of `counting_sheep_ott_comments.py`, use the # to write a comment explaining to yourself what the line does.

77.3 counting_sheep_floating.py

- Rewrite counting_sheep.py as counting_sheep_floating.py to use floating point numbers (hint: 10.0 is floating point).

77.4 float_10.py

What do you think the result of this code snippet will be?

```
print("float('10')", float('10'))
```

77.5 int_10.py

What do you think the result of this code snippet will be?

```
print(int(-10.9))
```

77.6 div_15_2.py

What is the output of the following?

```
print("15/2 = ", 15/2)
```

77.7 kmph_to_mph.py

Given that there are about 1.61 kilometres in a mile, if you jog 5 kilometres in 20 minutes and 22 seconds, what is your average speed in miles per hour? Call your program kmph_to_mph.py.(9.15 mph)

77.8 mileage.py

Write a short program to calculate the amount someone would expect for a work mileage rate of 45p for the first 10,000 miles and 20p thereafter in a year?

77.9 three_point_5.py

Which of the following give **3.5**? There are more than one.

```
print(7//2.0)
print(7/2)
print(7.0//2)
print(7.0/2.0)
```

77.10 x1_and_x2.py

Before running, predict what the output of this code will be?

```
x1 = 100
x2 = 3
x1 = x1 + 10
x1 = x1 + x2
x2 = x1
x2 = x2 + x1
print(x2)
```

77.11 remainder.py

What is the result of this **floor** division?

```
print("3.0//2.0 = ", 3.0//2.0)
```

77.12 remainder7.py

Which of the following give **7.0** as a result? There are more than one.

```
print(15//2.0)
print(15/2)
print(14/2)
print(15//2)
print(15.0//2)
```

CHAPTER

SEVENTYEIGHT

NUMPY AND MATRICES [NUMPY_LINEAR_EQUATIONS.RST]

Use NumPy to write programs to solve the system of equations:

```
x  + 2y = 5
3x + 4y = 6
```

78.1 numpy_solve_inv.py

This uses the `inv`(erse) and `dot` methods. This is only reliable if the system is **well** behaved.

```
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.array([5, 6])

x = np.linalg.inv(a).dot(b)

print(x)
print(a @ x)
```

78.2 numpy_solve.py

This uses the `solve` function and will be more reliable that the previous program.

```
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.array([5, 6])

x = np.linalg.solve(a, b)

print(x)
print(a @ x)
```

CHAPTER

SEVENTYNINE

NUMPY AND MATRICES [NUMPY_MATRIX_MULTIPLICATION.RST]

There are several ways to do matrix multiplication in NumPy. One is to use the @ operator. Try the following.

79.1 matrix_mul.py

```
import numpy as np

a = np.array([
    [1, 2],
    [3, 4]])

b = a @ a

print(b)
```

79.2 matrix_mul_trans.py

Comment this to explain what is going on.

```
import numpy as np

a = np.array([
    [1, 2],
    [3, 4]])

b = a.transpose()

c = a @ b

print(c)
```

79.3 matrix_det.py

From your maths classes, you might remember **matrix determinants**. Determinants are a way of characterising square matricies. You can use a Numpy (specifically the linear algebra package) to calculate determinants, however, the method below may not be suitable for large matricies (see the Numpy docs for more information).

```
import numpy as np

a = np.array(
    [[5, 2],
     [3, 4]])

print(np.linalg.det(a))
```

CHAPTER

EIGHTY

OPERATORS 1 AND THE INTERACTIVE PYTHON PROMPT [OP-01.RST]

Operators are special symbols for arithmetic or logical computation.

80.1 Mathematical operators

Here are some essential Python arithmetic operators.

`+` addition

`-` subtraction

`/` division

`*` multiplication

`//` floor division

80.2 Interactive arithmetic

Start Python in *interactive* mode. At your systems prompt (here >) type `python`. Recall, the >>> tells you that you are in Python *interactive* mode.

```
> python
>>>
```

Note:

- Do not guess the answers. At least one of them may not be what you expect.
- Do not type >>>. This is the prompt supplied by Python.

Type the following at the Python prompt (>>>). Note (type manually) the results in a file that you should name `interactive_results.txt` .

```
>>> 1.0 + 1
>>> 1 + 1
>>> 1 / 3
>>> 4 / 3
>>> 4 // 3
```

```
>>> 4.0 // 3
>>> 1 / 3.0
>>> 1 + 2 * 3
>>> (1 + 2) * 3
```

80.3 Logic 1 - relational operators

Boolean algebra is a branch of algebra related to the **truth**, and uses values *true* and *false*. Relational operators test the relationship between two entities. In Python, Boolean `True` or `False` are the result of relational operations. Here the list of Python relational operators.

> == check if values equal.
>
> != check if values are not equal.
>
> > check if the left value is greater than right.
>
> < check if the left value is less than right.
>
> >= check if the left value is greater than or equal to the right value.
>
> <= check if left value is less than or equal right value.

80.4 Logic 1 - relational operators exercise

Start Python in *interactive* mode.

```
> python
>>>
```

Type the following at the Python prompt (>>>) and note (manually) the results in your `interactive_results.txt` .

```
>>> 1 <= 1
>>> 1 < 1
>>> 1 >= 1
>>> 1 == 1
```

80.5 Quiz

Try to figure out the answer to each question **before** you run the code.

80.6 *Math*

Predict before checking the results of these sums.

1 + 1

2.0 + 1

2 * 3 + 1

(2 * 3) + 1

2.0 * (3 + 1)

80.7 Logic

Predict the result before checking these logic operations.

1 == 2

1 > 2

1 >= 2

1 <= 2

CHAPTER

EIGHTYONE

ADDITIONAL REVISION EXERCISES [OP-02.RST]

81.1 `Calculator 2

Predict before checking the results of these sums.

> 1 + 1
>
> 2.0 + 1
>
> 2/3 + 1
>
> 2/3.0 + 1
>
> 2 * 3 + 1
>
> (2 * 3) + 1
>
> 2 * (3 + 1)
>
> 1 == 2
>
> 1 = 2
>
> 1 > 2
>
> 1 >= 2
>
> 1 <= 2

CHAPTER

EIGHTYTWO

OPERATORS - MODULO (%) OPERATOR [OP-03-MODULO.RST]

The **modulo** (%) operator gives the **remainder** of a division of the left hand side by the right hand side of the %.

82.1 modulus.py

Write a commented program, modulus.py, to carry out the following calculations. You will want to experiment in interactive mode.

```
print(1 % 3)
print(90 - 25 * 3 % 4)
print(4 + 3 + 2 - 6 + 5 % 3 - 2 / 5 + 7)
```

82.2 modulus2.py

Predict before checking the results of these sums.

```
print(123 % 2)
print(1234 % 2)
```

CHAPTER

EIGHTYTHREE

PREAMBLE [PREAMBLE-01.RST]

Getting started.

83.1 Environment - shell/command line

- These notes assume Python version 3.
- Unless otherwise stated, these notes assume that you are using your computer's **shell** (command line input) to interact with Python and not an integrated development environment (**IDE**).
- **If you are not using a shell then modify these instructions appropriately.**
- In these notes, your system's **shell** is indicated by >, however, your system might have `$` or other symbol.

83.2 The interactive shell

There are several methods of interacting with Python. One, indicated by 3 chevrons (>>>), is Python's *interactive mode*. When in *interactive* mode the 3 chevrons tells you that Python is ready. Try the following:

1. On your system, from your command line type `python` and press `RETURN` to start Python in *interactive* mode.
   - `> python`
   - `>>>`
2. This is called the Python **shell**. Say something nice e.g.
   - `>>> print("Hi")`
   - Here, `print` is a Python function that displays your message, contained within brackets and quotes, to the output.
3. Exiting Python
   - To quit the Python interactive shell type `exit()` at the Python prompt then press **RETURN**.
   - `>>> exit()`
   - Alternatively, key `Ctrl-Z` to exit.

83.3 IPython shell

Another common **enhanced** Python **shell** is **IPython**. It is often the preferred *shell* used by scientists. If you have IPython installed, read its documentation to find out how it is launched on your computer. You could try on your command prompt:-

```
> ipython
```

Its interactive shell looks like this.

```
In [1]:
```

Display another friendly message.

```
In [1]: print("Hi")
```

CHAPTER

EIGHTYFOUR

SAVING YOUR WORK [PREAMBLE-015-SAVING.RST]

Create a directory to save your work.

- `/your_name_python_code`

As well as saving your work on your computer, at the end of the class, you should take copies with you or save on a network drive that you can access outside the class.

CHAPTER

EIGHTYFIVE

ENVIRONMENT - IDLE IDE [PREAMBLE-02.RST]

IDLE is a basic **IDE** (integrated develop environment) which *usually* comes as part of a default Python installation. IDLE is intended to be simple and suitable for beginners in an educational environment. However, do not expect too much from it and **do not** use it for GUI development.

85.1 Launch IDLE

Launch IDLE (we may need to discuss, or ask the Internet, how best to do this).

```
> idle3
```

85.2 Use the interactive shell

In the interactive **shell** (indicated by 3 chevrons, >>>) say something nice e.g.

```
>>> print("Hi")
```

85.3 Create a program from IDLE, `idle_01.py`

Select **File > New Window**.

In the new window (editor) select **File > Save as...** then `yourname_python_code/idle_01.py`.

In the editor write the following code.

```
message = "This was created using IDLE.\n"
print(10 * message)
```

Run the program, in the editor **Run > Run Module**.

CHAPTER

EIGHTYSIX

ENVIRONMENT - WINDOWS POWERSHELL PRACTICE [PREAMBLE-03.RST

A **shell** is a program that allows you to interact with your computer by inputting text. This is also commonly know as the **command** line. In the *Microsoft Window's* world, as a choice of **shell** program, I recommend the use of *Window's PowerShell*.

86.1 Run PowerShell

Depending on your system, you might run **PowerShell** from the *Windows start menu* of your PC by typing (do not type the >):

```
> powershell
```

86.2 PowerShell practice

Type the following in the **PowerShell** command prompt (do not type the >).

- To find out where you are type **pwd** (print working directory).

  ```
  > pwd
  ```

- Check the contents of a directory (list directory) with **ls**.

  ```
  > ls
  ```

- Make a new directory using **mkdir**.

  ```
  > mkdir new_directory_name
  ```

- Use **cd** (change directory) to move into a directory.

  ```
  > cd the_name_of_the_directory
  ```

- Checking with **pwd**, use *two dots* as a short hand to move up one directory.

  ```
  > cd ..
  ```

CHAPTER

EIGHTYSEVEN

ENVIRONMENT - CREATE DIRECTORY FOR YOUR CODE [PREAMBLE-04.RST]

Use *Windows* **PowerShell** to create a top-level directory for your Python code. This should be in a location that will be permanent. (We will discuss the best place on the system for this.)

```
> mkdir yourname_python_code
```

```
> cd yourname_python_code
```

CHAPTER

EIGHTYEIGHT

ENVIRONMENT - NOTEPAD++ INTRODUCTION [PREAMBLE-05.RST]

In most of this document, I will assume that you are not using an **IDE**. Instead, for writing your programs, I am assuming you are using a simple editor. For *Window's* users, I am recommending the use of **Notepad++**.

Using **Notepad++** (editor), create an empty text file **results.txt** as follows:

- Open **Notepad++**.
- Using **Save As...**, save your file in your `yourname_python_code/` directory as `results.txt`.
- Using *Windows Powershell*, verify that the file has been correctly created and in the correct place.

  > `pwd`

  `yourname_python_code/`

  > `ls`

  `results.txt`
- You will use this file to record your `results in a later section.

CHAPTER EIGHTYNINE

ENVIRONMENT - CREATE DIRECTORY HIERARCHY FOR YOUR CODE, DIRECTORIES [PREAMBLE-06.RST]

Continue creating directories until your *directory hierarchy* looks like this:

```
yourname_python_code/
containers/
control/
classes/
functions/
io/
numbers/
strings/
variables/
```

CHAPTER

NINETY

INTRODUCTION TO PYTHON PROGRAMMING - EXERCISE SHEET [PREAMBLE-07.RST]

90.1 Running

Which of these techniques can launch a Python script **script.py**?

1. c:> python script.py
2. Double click on the program icon.
3. Using python shell command line.
4. From within the an IDE.

CHAPTER

NINETYONE

INTRODUCTION TO PYTHON PROGRAMMING - EXERCISE SHEET 2 [PREAMBLE-08-EX_2.RST]

CHAPTER

NINETYTWO

WAYPOINT [PREAMBLE-10-WAYPOINT.RST]

Well done at getting this far through the work. Already, you know enough Python to put it to some practical use. In the following sections you will reinforce what you have learned, extend your knowledge and learn some new concepts. There will be repetition, however, do not skip any sections and go at a steady pace. Even if revision, it will still be excellent practice.

Try to figure out the answer to each question before you run the code. You do not have to do these questions from memory. Reference your notes, Internet and use the Python interactive prompt. There will be some new material. To understand the code you may need to carry out some of your own investigation, including on-line.

CHAPTER

NINETYTHREE

TKINTER MODULE [PROJECTS-01.RST]

Tkinter is a basic and limited GUI toolkit but has the advantage that it is usually installed at the same time as a standard Python installation.There are many powerful GUI tool kits that can be used with Python. If you are interested then search the internet for *python gui toolkits*.

93.1 hello_gui.py

Write and test the following code, `hello_gui.py`. I recommend that you run this from the terminal and not through a development environment. It is possible that an **IDE** (check your documentation) may encounter *threading* problems.

```
from sys import exit
from tkinter import *
root = Tk()
Button(root, text='Hello World!', command=exit).pack()
root.mainloop()
```

CHAPTER

NINETYFOUR

PROJECTS [PROJECTS-02.RST]

94.1 Project - files

- Go back to the program you wrote to collect class statistics. Improve it by adding persistence (saving data to a file) and allowing it to be updated.
- Improve your previous `details.py` program from the **dictionaries** section by saving and reading data from a file. This will require some internet research on how to **for** loop over a dictionary.
- Create a program, `upper_file.py`, that will changes all the contents of a given file into upper-case and saves them with new name.
- Reuse functions from your programs `fahrenheit_to_celsius.py` and `celsius_to_fahrenheit.py` (try to use `import` instead of copy and paste) to convert data files from one temperature scale to another.

CHAPTER

NINETYFIVE

PROJECT, OPTIONAL - IMAGE RESIZER, [THUMBNAILER] (SOLUTIONS/PROJECTS/THUMBNAILER.RST) [PROJECTS-03.RST]

Does not currently work.

- Write and test this code. You will have to find some images. One technique to run this program (on my computer) is to create a directory within the program's directory called **images/**, add several images and then:

  > python imageresizer.py images/*

- Optional improvements:
- Use function(s) to make the program potentially more reusable (`thumbnailer_funcs.py`).
- Make your code more flexible by allowing the `size = 128, 128` to be changed by user input.
- Use `if __name__ == "__main__":` to write some test or program control code.
- Create a GUI version. Instead of copying code from your existing program, use **import** to access functionality that you have already created (**thumbnailer_gui.py**).

```
# thumbnailer.py
# Credit: http://www.pythonware.com/library/pil/handbook/introduction.htm

import os
import sys
import Image

size = 128, 128

for infile in sys.argv[1:]:
    outfile = os.path.splitext(infile)[0] + ".thumbnail"
    if infile != outfile:
        try:
            im = Image.open(infile)
            im.thumbnail(size)
            im.save(outfile, "JPEG")
        except IOError:
            print "cannot create thumbnail for", infile
```

CHAPTER

NINETYSIX

LINE CONTINUATION [PROJECTS-04.RST]

Note the use of the *line continuation* character, \, to break up a long line. It was only used here to help with the print out of this document. There is no need for you to do the same. Just put the relevant code on one line.

CHAPTER

NINETYSEVEN

PROJECT - ARITHMETIC GAME [PROJECTS-05.RST]

Create a game, `arithmetic_game.py`. The game tests a players ability carry out arithmetic. Build its functionality and complexity steadily. Change the name of the program file as you add new functionality. (Hint, `import random`.)

Features could include:-

- a welcome
- adding two numbers, randomly selected
- scoring
- decrementing scoring
- gradually more difficult problems
- Bigger numbers
- Subtraction
- More than two numbers
- game levels
- Saving a high score
- two player game

CHAPTER

NINETYEIGHT

PROJECT - HEADS OR TAILS [PROJECTS-06.RST]

Your task is to write test and *improve* this program, **heads_or_tails.py**. Remember to build it up iteratively.

Covers:

- import
- random module
- functions
- standard output

Suggested improvements:-

- Use more functions.
- Use `if __name__ == "__main__":` to contain program control logic.
- Keep a score.

```
# heads_or_tails.py

from random import randint

def clear_screen():
    print 100*"\n"

def fun():
    print "'h': heads or 't': tails"
    print "'q': quit"
    print "\n"
    user_guess = input('CHANCER > ')
    clear_screen()
    return user_guess

clear_screen()
print "=================="
print "Welcome to CHANCER"
print "=================="
print "The excellent heads and tails game."
print "Hit the return key to begin."
input()
clear_screen()
```

(continues on next page)

(continued from previous page)

```
user_guess = fun()
while user_guess != 'q':
    user_guess = fun()
    if randint(0,1):
        print "HEADS"
        result = 'h'
    else:
        print "TAILS"
        result = 't'
    if user_guess == result:
        print "You are correct."
    else:
        print "You are wrong."
```

CHAPTER

NINETYNINE

PROJECT - *WORD TUMBLE* [PROJECTS-07.RST]

This is a fantastic game where the player has to make words from a set of letters they are given. In this game, a player types words made out of random letters that are at least 3 letters long. It was developed form an idea by Paul Hudson.

99.1 Game specification

Create a word game *Word Tumble* with specification:

- To launch program:

  > python word_tumble.py

- User is greeted with the message:

  Get ready to word tumble! Type 2 to exit, 1 for a new word.

- Select a random word from a word dictionary file **words.txt** (ask for a copy) and display with one space between each letter, for example:

  k n o c k o u t s

- The user inputs

  > knock

- The game replies with

  Good! Type 2 to exit, 1 for a new word.

  k n o c k o u t s

- You can keep going until the you exit (2) or select a new set of letters (1).
- Beware, don't use less than 3 letters, a word that does not exist or the same word twice. You will get messages like:

  The word has to have 3 or more letters

  The letters you used are not from the question

  You've used that already!

- Here are some suggestions on how you might code the game.
- You should build your version gradually, steadily testing and adding functionality.
- In the **:raw-latex:`\verb|words.txt|`** file, all the words have been set to lower case and words under 3 letters removed. I generated it using a program called Aspell. However, for development, before using this, create a short **:raw-latex:`\verb|list|`** of words for the program to use.

- The game has to keep going until the player has had enough. This requires a while loop.

```
play = True
while (play)
    pass
    # code goes here
```

- The game functionality should be above a main loop in the form of functions or classes.

CHAPTER

PROJECT - STOP WATCH AND REACTION TIMER [PROJECTS-08.RST]

Develop two related *time* projects in parallel. These should share some of the same code that you write. They will *import* some fundamental functions or classes from each other. To help reuse and sharing between your code projects, make your functions and classes small and focused on one task - do one thing well. This design is subjective and develops through experience. Your two projects are:

- Stop watch
- Reaction timer

The modules **time**, **datetime** and **calendar** modules may help.

After delivering a **simplestopwatch.py** and **reactiontimerone.py**, extend the functionality e.g.

- lap time
- lap time recall
- saving times
- scores
- levels
- GUI version

CHAPTER

ONE

PROJECT - FILES [PROJECTS-09.RST]

Write a program that reads input from the console, collects this in a list and finally saves this data to a file.

The following pseudo code and comments provides a framework for you program.

```
# while loop to read input from console
    # Append input to list
# end of while loop

# Create file object in write mode.

# Write lines to file object.

# Close file object.
```

CHAPTER

TWO

PROJECT - DICTIONARIES 2 - PERSONAL DATA PROGRAM [PROJECTS-10.RST]

The following, `details.py`, is a skeleton program designed for collecting personal data. For this exercise, use the **containers** directory.

```
# details.py

def read_from_terminal():
    user_input = input(
        '\n Return to continue or (q)uit and return: ')

    if (user_input == 'q'):
        return False
    else:
        personal_information_dict = {}
        personal_information_dict['first_name'] = \
            input('First name: ')
        personal_information_dict['age'] = \
            int(input('Age : '))
        return personal_information_dict


list_of_details = []
while(True):
    user_input = read_from_terminal()
    if user_input:
        list_of_details.append(user_input)
    else:
        break

print list_of_details
```

- Write and test the above, **details.py**.
- Add comments explaining its operation. To understand fully, you will have to do some internet research. In particular, this is the first time you have used **break**.
- Using this code as a starting point, improve the program, extend the dictionary and collect additional data such as:

  Second name

  Date of birth

Address

Email address

Sex

etc.

CHAPTER

THREE

PROJECT - COPY APPLICATION [PROJECTS-11.RST]

Write a program **copy_file.py** that

- takes two files names as command line arguments
- and copies the contents of one file to the other.
- Your program should check if the output file already exists (hint: **from os.path import exists**)
- and gives the user the choice to overwrite or quit.

103.1 Suggested solution

```
# copy_file.py

from sys import argv
from os.path import exists

program_name, original_file, copy_file = argv

print "Copying from %s to %s" % (original_file, copy_file)

from_file = open(original_file)
data = from_file.read()

print "Output file exist? %r" % exists(copy_file)
print "RETURN to continue or CTRL-C to quit."
input('?')

to_file = open(copy_file, 'w')
to_file.write(data)

from_file.close()
to_file.close()
```

CHAPTER

FOUR

PROJECT - CANOPY IDE WITH A PLOTTING EXAMPLE # [PROJECTS-12.RST]

There many different *modules* that provide data plotting functionality. **Canopy** is a scientific Python environment that by default comes with plotting *modules* and makes installing others easy.

- Open *Canopy* and select *Editor*.
- Interactive plotting. Write the following in the interactive shell.

  In[]: data = [1, 2, 3, 4, 5, 6]

  In[]: plot(data)
- Close the figure window.

104.1 canopy_plot_01.py

Write the following `canopy_plot_01.py` in the editor.

```
# canopy_plot_01.py
# http://matplotlib.org/users/pyplot_tutorial.html

import numpy as np
import matplotlib.pyplot as plt

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

CHAPTER

FIVE

PROJECT - PLOTTING EXAMPLE [PROJECTS-13.RST]

105.1 plot_heat.py

Write and test this code.

```
import numpy as np
import matplotlib.pyplot as plt

# Generate some test data
x = np.random.randn(8873)
y = np.random.randn(8873)

heatmap, xedges, yedges = np.histogram2d(x, y, bins=50)
extent = [xedges[0], xedges[-1], yedges[0], yedges[-1]]

plt.clf()
plt.imshow(heatmap, extent=extent)
plt.show()
```

Original code

http://stackoverflow.com/questions/2369492/generate-a-heatmap-in-matplotlib-using-a-scatter-data-set

105.2 plot_heat_cmts.py

With the help of the investigate, instigate and comment your previous code and save it as `plot_heat_cmts.py`.

CHAPTER

SIX

PLOT_ANNIMATION.PY [PROJECTS-13-1.RST]

Write and test this code.

(Original code http://matplotlib.org/examples/animation/basic_example.html.)

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def update_line(num, data, line):
    line.set_data(data[...,:num])
    return line,

fig1 = plt.figure()

data = np.random.rand(2, 25)
line_plot_list, = plt.plot([], [], 'r-')
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.xlabel('x')
plt.title('test')
line_ani = animation.FuncAnimation(fig1, update_line, 25
    , fargs=(data, line_plot_list), interval=50, blit=True)
plt.show()
```

106.1 plot_annimation_comments.py

In the previous code there are some concepts that are beyond this introductory course, however, do your best to comment and save as version as `plot_annimation_comments.py`.

CHAPTER

SEVEN

NUMERICAL [PROJECTS-14-TIME_MATRIX_SOLVE.RST]

With reference to the Numerical lecture from Oxford that I have downloaded.

2015-10-13-MT_scientific_computing_lecture_01.mp4

```
%matplotlib inline
import numpy as np
import pandas as pd

times=[]
n_cubed = []
n = []

for i in range(1, 13):
    c = 2**i
    r = c
    A = np.random.random((r, c))
    b = np.ones((r, 1))
    #x = np.linalg.solve(A, b)
    t = %timeit -o -n1 -r1 np.linalg.solve(A, b)
    n_cubed.append(c**3)
    times.append(t)
    n.append(c)

t = [t.best for t in times]

df = pd.DataFrame({"n^3":n_cubed, "t":t, "n":n})

df["n^3"] = df["n^3"]/5500000000

df.plot(x="n", y=["n^3", "t"])


((2**13)**3)/5500000000

n_cubed
```

```
%matplotlib inline
import numpy as np
import pandas as pd
import timeit as ti
```

(continues on next page)

(continued from previous page)

```
times=[]
n_cubed = []
n = []

for i in range(1, 13):
    c = 2**i
    r = c
    A = np.random.random((r, c))
    b = np.ones((r, 1))
    #x = np.linalg.solve(A, b)
    t = ti.timeit(stmt="np.linalg.solve(A, b)", number=1, globals={"np":np, "A":A, "b":b}
↪)
    n_cubed.append(c**3)
    times.append(t)
    n.append(c)

#t = [t.best for t in times]

t = times

df = pd.DataFrame({"n^3":n_cubed, "t":t, "n":n})

df["n^3"] = df["n^3"]/5500000000

df.plot(x="n", y=["n^3", "t"])


((2**13)**3)/5500000000

n_cubed
```

CHAPTER

EIGHT

[PROJECTS-15-HOLVIEWS.RST]

conda install -c pyviz holoviews

http://holoviews.org/user_guide/Responding_to_Events.html

```
import numpy as np
import holoviews as hv
from holoviews.streams import Stream, param
hv.extension('bokeh', 'matplotlib')
```

```
def sample_distributions(samples=10, tol=0.04):
    np.random.seed(42)
    while True:
        gauss1 = np.random.normal(size=samples)
        gauss2 = np.random.normal(size=samples)
        data = (['A']*samples + ['B']*samples, np.hstack([gauss1, gauss2]))
        yield hv.BoxWhisker(data, 'Group', 'Value')
        samples+=1

sample_generator = sample_distributions()
```

```
dmap = hv.DynamicMap(sample_generator, streams=[Stream.define('Next')()])
```

```
dmap.periodic(0.1, 1000, timeout=3)
```

```
dmap
```

CHAPTER

NINE

NUMPY [PROJECTS-16-NUMPY.RST]

NumPy is a package for scientific computing with Python. NumPy usually operates on n-dimensional arrays with elementwise operations. Thus when you multiply a Numpy array using the * operator it is **not** mathematical matrix multiplication.

Experiment with the following in interactive mode.

109.1 Import Numpy

```
>>> import numpy as np
```

109.2 Adding two arrays

```
>>> a = np.array([2,3,4])
>>> b = np.array([1,2,3])
>>> a + b
```

109.3 Raise to the power of 3

```
>>> a**3
```

109.4 Create an array and reshape it

```
>>> a = np.arange(10).reshape(2, 5)
>>> a
```

109.5 Check its shape

```
>>> a.shape
```

109.6 How many dimensions

```
>>> a.ndim
```

109.7 What is the type of the array elements

```
>>> a.dtype.name
```

109.8 Number of elements of the array

```
>>> a.size
```

109.9 What is the type of the array

```
>>> type(a)
```

CHAPTER

[PROJECTS-17-AST.RST]

https://docs.python.org/3/library/ast.html

https://en.wikipedia.org/wiki/Abstract_syntax_tree

https://stackoverflow.com/questions/15197673/using-pythons-eval-vs-ast-literal-eval

https://www.mattlayman.com/blog/2018/decipher-python-ast/

CHAPTER

ONE

[PROJECTS-18-PANDAS.RST]

https://towardsdatascience.com/a-quick-introduction-to-the-pandas-python-library-f1b678f34673

111.1 Pandas - interactive

Pandas is library written for data wrangling and analysis. Pandas is most appropriate when all your data can fit into memory. The name **Pandas** comes from **panel data**, an econometrics lingo for observations over time.

Pandas comes with

- data structures
- data operations
- data manipulating
- numerical tables
- time series

111.2 How to import pandas

- Importing a library brings it into memory for you to use.
- By convention most people use the `as pd` alias.

```
import pandas as pd
```

111.3 Create and save CSV data using pandas

It can take raw data from disk (like a CSV or a SQL database) and create a **data frame** - a Python object with rows and columns, like a spreadsheet.

111.4 Converting to a Pandas `DataFrame`

```
pd.DataFrame
```

111.5 Series

111.6 Opening a data file using Pandas

111.7 Open a remote file using Pandas

111.8 Viewing and inspecting the data

111.9 head

111.10 tail

111.11 shape

111.12 info

111.13 value_count

111.14 Statistics

111.15 describe

111.16 mean

111.17 corr

111.18 count

111.19 max

111.20 min

111.21 median

111.22 std

111.23 Selecting data

111.24 Column

111.25 Row by position

111.25.1 Select element

CHAPTER

TWO

SORT

112.1 `groupby`

112.2 Wrangling

CHAPTER THREE

QUIZ - LEARNING AND REVISION

You do not have to do these questions from memory. Reference your notes, Internet and use the Python interactive prompt. There will be some new material. To understand the code you may need to carry out some of your own investigation, including on-line.

CHAPTER

FOUR

EXERCISES 1

114.1 Script mode, strings, comments and errors [string-01-script_mode.rst]

In Python's **interactive** mode, when you exit you lose your work. In this exercise, instead, you will use Python's **script** mode to run your pre-written programs. You can then re-run your programs as many times as you like.

114.1.1 `hello_world.py`

1. Using an editor, type the following text into a single file named `hello_world.py`. Note, Python will work better with files ending in `.py`.

   ```
   print("Hello World!")
   ```

2. The directory path and file name, under your directory, should be `/your_name_python_code/hello_world.py`.
3. Run the program. In the terminal, in the same directory as the file, type

   ```
   > python hello_world.py
   ```

114.1.2 `hello_world_extra_line.py`

1. Copy all your previous `hello_world.py` code into a new file, `hello_world_extra_line.py`.
2. Have your new program `print` another line of your own at the end.

114.1.3 Comments - `comment_line_hello_world.py`

Make a program that prints only the first line of `hello_world_extra_line.py` as follows:

1. Copy all the code from `hello_world_extra_line.py` to `comment_line_hello_world.py`.
2. Put a '`#`' character at the beginning of the second line.
3. Run your program.
4. What did the '`#`' do?

114.1.4 Comments - comments_program.py

Write and test the following program, comments_program.py.

```
# This is a single line comment.
# A comment can be used to add useful information.
# Anything after the `#` is ignored by python.

print("Comment at the end of a line.") # ignored
# A comment can "disable" or "comment out" a piece of code.
# print("This will not run.")
print("This will run normally.")
# Note, normally you only write comments to explain *difficult to understand* code.
```

114.1.5 Strings 1 - using quotes, quotes.py

In this exercise you will discover some handy tricks for displaying text.

- Save the following into a file, named quotes.py, i.e. /your_name_python_code/quotes.py.

You should build your program up in stages, testing as you go along - *iteratively*.

```
print("As well as interactive mode, you can save your code in a file - 'script'.")
print("Conventionally, Python programs have names that end in '.py'.")
print('Python is a "high-level" programming language.')
print("Computers can only run 'low-level' program languages.")
print('"High-level" languages have to be processed (compiled) into "low-level"')
print("before they can run.")
print('High-level languages are easier to program and more portable.')
print("Python reads, translates and runs one line at time, an 'interpreted'")
print("language, and it's not a compiled language, that first translates all your")
print('code before running.')
```

- Run this program, **not** at the Python prompt (>>>) but from your computers **command line** terminal prompt, thus

```
> python quotes.py
```

114.1.6 String operations, string_math_operations.py

In this exercise you are going to discover that it is possible to use some maths operations on strings.

Write and test the following program string_math_operations.py.

```
print("University of Python")
print("University of Python" * 3)
print("University of Python" + " " + "University of Python")
```

114.1.7 string_math_operations_comments.py

- Write a comment above each line of string_math_operations.py explaining what is going on.
- Save this as string_math_operations_comments.py.

114.1.8 string_math_operations_one_hundred_times.py

Write a program, string_math_operations_one_hundred_times.py, that displays the word Python 100 times.

114.1.9 Errors 1 - syntax error and print

Programming errors are called **bugs** and tracking them down and fixing them is called **debugging**. If you break the rules that define the structure of Python you create a **bug** called a **syntax error**.

114.1.10 syntax_error.py

When you are using print to display a message, the message must be contained within the same kind of quotes (two sets of "double" or two sets of 'single' quotes). Test this by writing the following program, syntax_error.py .

```
print("This is an error')
```

When you run, you should get a response like:

```
File "<ipython-input-2-b5ed948c50bb>", line 1
    print("This is a an error')
                              ^
SyntaxError: EOL while scanning string literal
```

114.1.11 python_is_easy.py

Write a trivial Python program, python_is_easy.py, that gives the message:

```
Python is easy.
'When you know how.'
And, I "know"!
```

114.1.12 hello_three.py

This program should print **Hello** six times. Fix it by making one change and save as hello_six.py.

```
print('Hello Hello Hello')
print('Hello ' + 3)
```

114.2 Variables, errors and type coercion [var-01.rst]

114.2.1 Variables 1 - variables and iterative development

In this section you will find out about the use of variables in Python.

A variable is a name that refers to a value.

- You can think of a **variable** as a label to data that you want to reuse.
- The *assignment* (equals sign, =) operator is used to create a statement that allocates a variable name to a **value**.

Variable names can contain both letters and digits but they must begin with a letter or underscore. By convention variable names do not start with a capital letter.

114.2.2 `chocs.py`

Iteratively (write and testing one line at a time) create this program, `chocs.py`.

- Write the line

  `number_of_chocs = 500`

- Run the program and fix any errors.

  `> python chocs.py`

- Write the next line

  `space_in_a_belly = 4.0`

- Again, run the program and fix any errors.

  `> python chocs.py`

Repeat this iterative, write and test, process until you have the complete the following program.

```
# chocs.py

number_of_chocs = 500
space_in_a_belly = 4.0
number_of_people = 120
number_of_dieters = 30

greedy_guts = number_of_people - number_of_dieters

chocolate_capacity = greedy_guts * space_in_a_belly

left_over_chocs = number_of_chocs - chocolate_capacity

chocs_per_doggy_bag = left_over_chocs / number_of_people

print("There are", number_of_chocs, "chocolates available")
print("Approximately", chocolate_capacity, "will be devoured.")
print("There will be", left_over_chocs, "left over.")
print("Which is ", chocs_per_doggy_bag, " each to take home.")
```

114.2.3 chocs_ott_comments.py

Write a new version of the previous chocs.py code, named chocs_ott_comments.py, but with a comment above each line explaining what it does.

114.2.4 Errors 2 - chocs_broken.py

Saving as chocs_broken.py, break your previous chocs.py code to produce the error:

```
Traceback (most recent call last):
...
greedy_guts = number_of_people - number_of_dieters
NameError: name 'number_of_dieters' is not defined
```

Note, if you are using an IDE to run your programs then make sure that the memory is cleared between runs. Otherwise, you might not be able to create this error. This may require a configuration setting change.

114.2.5 Variables 2.3 - coerce type, chocs_integer.py

- In chocs.py change 4.0 to 4 and save as chocs_integer.py .
- Run the code and explain, in code comments, what is going on?

CHAPTER

FIVE

EXERCISES 2

115.1 Strings - embedding quotes [string-02.rst]

Strings are a series of characters contained in quotes. Double or single quotes are equivalent and can be used in combination to embed in the other. (Alternatively, to embed quotation marks inside strings you can use the escape \ character.)

115.1.1 burns.py

Demonstrate that you understand the embedding and escaping of quotes by writing and commenting each line of `burns.py`.

```
print("O my Luve's like a red, red rose,")
print("That's newly sprung in June:")
print('O my Luve\'s like the melodie,')
print('That\'s sweetly play\'d in tune.')
print(5 * "-")
print('By "Rabbie" Burns')
```

115.2 Variables 3 - variables and ``*type*`` conversions [var-03.rst]

There are times when you will need to convert between data types. You can achieve this with built in functions like **int(), long() and float()**. Use these to convert *strings* input by a user into a numbers. However, you should take care as information can be lost.

115.2.1 one_int_type.py

This program uses `type` and `int` *built-in* functions. Write this `one_int_type.py` program and try to guess the output before you run the code.

```
a = 1
b = '1'
c = 1.0
print(a, b, c)
print(type(a))
print(type(b))
print(type(c))
```

(continues on next page)

(continued from previous page)

```
d = int(b)
e = int(c)

print(d, type(d))
print(e, type(e))
```

115.2.2 one_int_type_ott_comments.py

Saving as `one_int_type_ott_comments.py`, copy and comment the previous `one_int_type.py`.

115.2.3 five_type.py

Create a program `five_type.py` that:

- Allocates the integer `5` to variable name `five_int`.
- Allocates the floating point number `5` to the variable name `five_float`.
- Allocates the string `5` to the variable name `five_string`.
- By printing the type, proves that you have allocated the the types correctly.
- Using the built in function `float`, multiply `five_int`, `five_float` and `five_string` to give `125.0`.

115.2.4 ten_convert.py

Use your knowledge of conversions to fix the program **ten_bad.py**. Save your program as **ten_convert.py**.

```
# ten_bad.py

print(10, '10', 10.0)
print(type(10))
print(type('10'))
print(type(10.0))
print(10/10.0, type(10/10.0))
print(10 * '10', type(10 * '10'))
print(10 / '10', type(10 / '10'))
```

115.2.5 ten_convert_ott_comments.py

Copy `ten_convert.py` fully comment and save as `ten_convert_ott_comments.py`.

115.3 input 3 and Conversion 2 - conversions and input [type-01.rst]

115.3.1 convert_broken.py

Write and test this `convert_broken.py`. It should fail with an error message like

> `TypeError: cannot concatenate 'str' and 'int' objects`

```
print("What is your first name?",)
name = input()
surname = input("What is your surname?",)
print("How old are you?",)
age = age + 1
print("In one year %s %s will be %d years old." % (name, surname, age))
```

115.3.2 convert_fixed.py

Copy your previous broken code `convert_broken.py` and, saving as `convert_fixed.py` , make a change so that it now works. (Hint `age = int(age_str)`)

115.3.3 convert_ott_comments.py

Copy the working code `convert_fixed.py` into `convert_ott_comments.py`. With comments, clearly explain why this works.

CHAPTER

SIX

EXERCISES 3

116.1 Strings [string-03.rst]

116.1.1 doing.py

Write and test the following program.

```
string_variable = "What am I doing?\n"
print(5 * string_variable)
```

116.1.2 doing_comments.py

Copy `doing.py`, thoroughly comment and save as `doing_comments.py`.

116.1.3 doing_badly.py

Copy `doing.py`, saving as `doing_badly.py` remove the `\n`, run and explain in comments what is going on.

116.1.4 Functions - parameters, functions_and_strings.py

Write and test the following program, `functions_and_strings.py`.

```
def function_two(argument1, argument2):
    print("argument 1: %s, argument 2: %s" % (argument1, argument2))

def function_one(argument):
    print("argument: %s" % argument)

def function_nothing():
    print("Zero.")

function_two("Two","One")
function_one("One")
function_nothing()
```

116.2 Numbers [var-04.rst]

116.2.1 pounds_dirham_convert.py

Write program of only a few lines to convert British Pounds into Moroccan Dirham and vice versa.

116.2.2 convert1234.py

Write a short program, of only a few lines, that converts the floating point number **123.4** into an integer.

116.2.3 Operators - numerical average_of_four.py

- Write a very short program, only a few lines, to calculate the average of 5, 102, -12.2 and 5, `average_of_four.py`.

116.3 Strings - r, another method, avoiding \, to escape quotes [string-06-raw_strings.rst]

Using **raw strings**.

- String literals may be prefixed by `r` or `R`.
- The string is then a **raw string**.
- **Raw strings** have different rules for escape characters.

116.3.1 string_escape_r.py

Write and test this program.

```
print("C:\python\number")
print(r"C:\python\number")
```

116.3.2 string_escape_r_comment.py

Make a copy of the previous program and write code `comments` explaining what is going on.

116.4 SymPy[sympy.rst]

Symbolic computation concerns calculations using mathematical symbolically rather than numbers. The results are exact and remain in symbolic form until they are forced into a numerical representation.

116.4.1 math module

Try the following in interactive mode.

```
import math
```

```
math.sqrt(2)
```

```
2 * math.sqrt(2)
```

```
math.sqrt(8)
```

116.4.2 sympy module

Try the following in interactive mode.

```
import sympy
```

```
sympy.sqrt(2)
```

```
2 * sympy.sqrt(2)
```

```
sympy.sqrt(8)
```

116.4.3 sympy display

Again, in interactive mode, type the following.

```
sympy.init_printing(use_unicode=True)
```

```
sympy.sqrt(2)
```

```
2 * sympy.sqrt(2)
```

```
sympy.sqrt(8)
```

CHAPTER

SEVEN

STRINGS 3.1 [STRING-04.RST]

Strings have many useful methods which are accessed through **dot** notation. To find out more search the internet for **python string methods**.

117.1 alphabet_upper.py

Write this program which uses the *upper* string method.

```
# alphabet_upper.py

alphabet_string = "abcdefghijklmnopqrstuvwxyz"
print(alphabet_string.upper())
```

117.2 my_name_caps.py

Write a program `my_name_caps.py`, that uses `alphabet_string = "abcdefghijklmnopqrstuvwxyz"` to write your name. It must give the correct capitalisation at the start of your name.

117.3 better_code.py

- You must only use the following separate strings:-

  `" b e t t e r "`

  `` “ code “``

  `` “ I “``

  `` “must “``

  `` “ w r i t e “``

  `` “ “``

  `` “ “``

  `` “ “``

  `` “ “``

- Write a program `better_code.py` that prints out the line:

```
I must write better code
```

CHAPTER

EIGHT

STRING FORMATTING [STRING-05-FORMATTING.RST]

118.1 Strings - string formatting float_formatting.py

Write, test and explain with comments float_formatting.py.

```
# float_formatting.py

print("%f" % 2.12345678)
print("%d" % 2.12345678)
print("%.2f" % 2.12345678)
```

118.2 beans_and_toast.py

Write and test the following program, demonstrating integers, functions, and formatting, beans_and_toast.py.

```
# beans_and_toast.py

def beans_n_toast(beans, toast):
    print("There are %d beans." % beans)
    print("There are %d slices of toast." % toast)
    print("\n")

beans_n_toast(200, 4)
number_of_beans = 100
numbers_of_toast = 3
beans_n_toast(number_of_beans, numbers_of_toast)
beans_n_toast(100 + 150, 2 + 6)
beans_n_toast(number_of_beans + 100, numbers_of_toast + 5)
```

118.3 beans_and_toast_ott_comments.py

Make a copy of previous code and save as `beans_and_toast_ott_comments.py`. Write a comment above each line explaining what is does.

CHAPTER

NINE

VARIABLES 2 [VAR-02.RST]

119.1 variables_a_b.py

Predict the output of the code fragment before testing.

```
a = 1
b = 2
b = a
print(a - b)
```

119.2 var_1_2.py

- Variables: `var1` and `var2`.
- Assign 10 to a variable, `var1`, and 20 to `var2`. Display the result of adding `var1` and `var2`.

CHAPTER

EXERCISES 1

120.1 Preamble [preamble-01.rst]

Getting started.

120.1.1 Environment - shell/command line

- These notes assume Python version 3.
- Unless otherwise stated, these notes assume that you are using your computer's **shell** (command line input) to interact with Python and not an integrated development environment (**IDE**).
- **If you are not using a shell then modify these instructions appropriately.**
- In these notes, your system's **shell** is indicated by >, however, your system might have `$` or other symbol.

120.1.2 The interactive shell

There are several methods of interacting with Python. One, indicated by 3 chevrons (>>>), is Python's *interactive mode*. When in *interactive* mode the 3 chevrons tells you that Python is ready. Try the following:

1. On your system, from your command line type `python` and press `RETURN` to start Python in *interactive* mode.
   - `> python`
   - `>>>`
2. This is called the Python **shell**. Say something nice e.g.
   - `>>> print("Hi")`
   - Here, `print` is a Python function that displays your message, contained within brackets and quotes, to the output.
3. Exiting Python
   - To quit the Python interactive shell type `exit()` at the Python prompt then press **RETURN**.
   - `>>> exit()`
   - Alternatively, key `Ctrl-Z` to exit.

120.1.3 IPython shell

Another common **enhanced** Python **shell** is **IPython**. It is often the preferred *shell* used by scientists. If you have IPython installed, read its documentation to find out how it is launched on your computer. You could try on your command prompt:-

```
> ipython
```

Its interactive shell looks like this.

```
In [1]:
```

Display another friendly message.

```
In [1]: print("Hi")
```

120.1.4 Saving your work [preamble-015-saving.rst]

Create a directory to save your work.

- /your_name_python_code

As well as saving your work on your computer, at the end of the class, you should take copies with you or save on a network drive that you can access outside the class.

120.2 Operators 1 and the interactive Python prompt [op-01.rst]

Operators are special symbols for arithmetic or logical computation.

120.2.1 Mathematical operators

Here are some essential Python arithmetic operators.

+ addition

- subtraction

/ division

* multiplication

// floor division

120.2.2 Interactive arithmetic

Start Python in *interactive* mode. At your systems prompt (here >) type `python`. Recall, the >>> tells you that you are in Python *interactive* mode.

> python

>>>

Note:

- Do not guess the answers. At least one of them may not be what you expect.

- Do not type >>>. This is the prompt supplied by Python.

Type the following at the Python prompt (>>>). Note (type manually) the results in a file that you should name `interactive_results.txt` .

```
>>> 1.0 + 1

>>> 1 + 1

>>> 1 / 3

>>> 4 / 3

>>> 4 // 3

>>> 4.0 // 3

>>> 1 / 3.0

>>> 1 + 2 * 3

>>> (1 + 2) * 3
```

120.2.3 Logic 1 - relational operators

Boolean algebra is a branch of algebra related to the **truth**, and uses values *true* and *false*. Relational operators test the relationship between two entities. In Python, Boolean `True` or `False` are the result of relational operations. Here the list of Python relational operators.

== check if values equal.

!= check if values are not equal.

> check if the left value is greater than right.

< check if the left value is less than right.

>= check if the left value is greater than or equal to the right value.

<= check if left value is less than or equal right value.

120.2.4 Logic 1 - relational operators exercise

Start Python in *interactive* mode.

```
> python

>>>
```

Type the following at the Python prompt (`>>>`) and note (manually) the results in your `interactive_results.txt` .

```
>>> 1 <= 1

>>> 1 < 1

>>> 1 >= 1

>>> 1 == 1
```

120.2.5 Quiz

Try to figure out the answer to each question **before** you run the code.

120.2.6 *Math*

Predict before checking the results of these sums.

> 1 + 1
>
> 2.0 + 1
>
> 2 * 3 + 1
>
> (2 * 3) + 1
>
> 2.0 * (3 + 1)

120.2.7 Logic

Predict the result before checking these logic operations.

> 1 == 2
>
> 1 > 2
>
> 1 >= 2
>
> 1 <= 2

120.3 Function calling and help [help-01-help.rst]

120.3.1 Functions 1 - calling

A function is a technique for grouping and reusing code. It is a predefined sequence of statements which can be *called* to carry out a computation. Functions often *take* arguments and *return* results. Python comes with some **built-in** functions. To **call** a function you use its name and parentheses - curved brackets. Indeed, you have already encountered Python's `print` function.

`help()`, is another example of a **built-in** function.

120.3.2 Help 1 - `help` function and interactive help

Including on-line documentation, tutorials, forums and printed books, there are many ways to find Python help. If you have a Python problem, as it is likely that someone else has already had a similar issue, your **first port of call is an internet search engine**. However, there is also interactive help built into your Python session. (Note, interactive help is not always installed.) In this section you will be investigating Python's built in help function.

Note for later. For `help` to work on a *statement* the *statement* needs to be in quotes. This is an aside and detail that you can put at the back of your mind.

120.3.3 Interactive help

Try this in the **Python** (>>>) terminal.

```
>>> help()
help>
    ...
```

Type `q` and `return` to quit interactive `help`.

```
help>q
```

To find help on a particular topic, e.g. `print`:

```
>>> help(print)
```

Help on an type or object, e.g. `1.1`:

```
>>> help(1.1)
```

120.4 Script mode, strings, comments and errors [string-01-script_mode.rst]

In Python's **interactive** mode, when you exit you lose your work. In this exercise, instead, you will use Python's **script** mode to run your pre-written programs. You can then re-run your programs as many times as you like.

120.4.1 `hello_world.py`

1. Using an editor, type the following text into a single file named `hello_world.py`. Note, Python will work better with files ending in `.py`.

   ```
   print("Hello World!")
   ```

2. The directory path and file name, under your directory, should be `/your_name_python_code/hello_world.py`.

3. Run the program. In the terminal, in the same directory as the file, type

   ```
   > python hello_world.py
   ```

120.4.2 `hello_world_extra_line.py`

1. Copy all your previous `hello_world.py` code into a new file, `hello_world_extra_line.py`.

2. Have your new program `print` another line of your own at the end.

120.4.3 Comments - `comment_line_hello_world.py`

Make a program that prints only the first line of `hello_world_extra_line.py` as follows:

1. Copy all the code from `hello_world_extra_line.py` to `comment_line_hello_world.py`.
2. Put a '#' character at the beginning of the second line.
3. Run your program.
4. What did the '#' do?

120.4.4 Comments - `comments_program.py`

Write and test the following program, `comments_program.py`.

```
# This is a single line comment.
# A comment can be used to add useful information.
# Anything after the `#` is ignored by python.

print("Comment at the end of a line.") # ignored
# A comment can "disable" or "comment out" a piece of code.
# print("This will not run.")
print("This will run normally.")
# Note, normally you only write comments to explain *difficult to understand* code.
```

120.4.5 Strings 1 - using quotes, `quotes.py`

In this exercise you will discover some handy tricks for displaying text.

- Save the following into a file, named `quotes.py`, i.e. `/your_name_python_code/quotes.py`.

You should build your program up in stages, testing as you go along - *iteratively*.

```
print("As well as interactive mode, you can save your code in a file - 'script'.")
print("Conventionally, Python programs have names that end in '.py'.")
print('Python is a "high-level" programming language.')
print("Computers can only run 'low-level' program languages.")
print('"High-level" languages have to be processed (compiled) into "low-level"')
print("before they can run.")
print('High-level languages are easier to program and more portable.')
print("Python reads, translates and runs one line at time, an 'interpreted'")
print("language, and it's not a compiled language, that first translates all your")
print('code before running.')
```

- Run this program, **not** at the Python prompt (>>>) but from your computers **command line** terminal prompt, thus

  ```
  > python quotes.py
  ```

120.4.6 String operations, `string_math_operations.py`

In this exercise you are going to discover that it is possible to use some maths operations on strings.

Write and test the following program `string_math_operations.py`.

```
print("University of Python")
print("University of Python" * 3)
print("University of Python" + " " + "University of Python")
```

120.4.7 `string_math_operations_comments.py`

- Write a comment above each line of `string_math_operations.py` explaining what is going on.
- Save this as `string_math_operations_comments.py`.

120.4.8 `string_math_operations_one_hundred_times.py`

Write a program, `string_math_operations_one_hundred_times.py`, that displays the word `Python` 100 times.

120.4.9 Errors 1 - syntax error and `print`

Programming errors are called **bugs** and tracking them down and fixing them is called **debugging**. If you break the rules that define the structure of Python you create a **bug** called a **syntax error**.

120.4.10 `syntax_error.py`

When you are using `print` to display a message, the message must be contained within the same kind of quotes (two sets of `"double"` or two sets of `'single'` quotes). Test this by writing the following program, `syntax_error.py` .

```
print("This is an error')
```

When you run, you should get a response like:

```
File "<ipython-input-2-b5ed948c50bb>", line 1
    print("This is a an error')
                               ^
SyntaxError: EOL while scanning string literal
```

120.4.11 `python_is_easy.py`

Write a trivial Python program, `python_is_easy.py`, that gives the message:

```
Python is easy.
'When you know how.'
And, I "know"!
```

120.4.12 hello_three.py

This program should print **Hello** six times. Fix it by making one change and save as `hello_six.py`.

```
print('Hello Hello Hello')
print('Hello ' + 3)
```

120.5 Variables, errors and type coercion [var-01.rst]

120.5.1 Variables 1 - variables and iterative development

In this section you will find out about the use of variables in Python.

A variable is a name that refers to a value.

- You can think of a **variable** as a label to data that you want to reuse.
- The *assignment* (equals sign, =) operator is used to create a statement that allocates a variable name to a **value**.

Variable names can contain both letters and digits but they must begin with a letter or underscore. By convention variable names do not start with a capital letter.

120.5.2 chocs.py

Iteratively (write and testing one line at a time) create this program, `chocs.py`.

- Write the line

  `number_of_chocs = 500`

- Run the program and fix any errors.

  `> python chocs.py`

- Write the next line

  `space_in_a_belly = 4.0`

- Again, run the program and fix any errors.

  `> python chocs.py`

Repeat this iterative, write and test, process until you have the complete the following program.

```
# chocs.py

number_of_chocs = 500
space_in_a_belly = 4.0
number_of_people = 120
number_of_dieters = 30

greedy_guts = number_of_people - number_of_dieters

chocolate_capacity = greedy_guts * space_in_a_belly

left_over_chocs = number_of_chocs - chocolate_capacity
```

(continues on next page)

(continued from previous page)

```
chocs_per_doggy_bag = left_over_chocs / number_of_people

print("There are", number_of_chocs, "chocolates available")
print("Approximately", chocolate_capacity, "will be devoured.")
print("There will be", left_over_chocs, "left over.")
print("Which is ", chocs_per_doggy_bag, " each to take home.")
```

120.5.3 chocs_ott_comments.py

Write a new version of the previous chocs.py code, named chocs_ott_comments.py, but with a comment above each line explaining what it does.

120.5.4 Errors 2 - chocs_broken.py

Saving as chocs_broken.py, break your previous chocs.py code to produce the error:

```
Traceback (most recent call last):
...
greedy_guts = number_of_people - number_of_dieters
NameError: name 'number_of_dieters' is not defined
```

Note, if you are using an IDE to run your programs then make sure that the memory is cleared between runs. Otherwise, you might not be able to create this error. This may require a configuration setting change.

120.5.5 Variables 2.3 - coerce type, chocs_integer.py

- In chocs.py change 4.0 to 4 and save as chocs_integer.py .
- Run the code and explain, in code comments, what is going on?

120.6 Logic operator v assignment operator [if-01-relational.rst]

120.6.1 Logic 2, relational operators == and !=, and assignment =

Relational operations result in either True or False.

- Gotcha, keep in mind that = is an **assignment** operator and == is a **relational** operator.
- The following exerciser uses the operator == to compare the objects on either side - True if they are equal.
- != compares the objects on either side - True if they are not equal.

120.6.2 relational_01.py

Solve these logic problems, first by hand then verify your results by running the program.

```
print(1 + 1 == 2)
print(1 + 1 == 3)
print(1 + 1 != 3)
print('right' == 'left')
print('right' == 'right')

a = 1 + 1
b = 2
print(a == b)
```

120.6.3 relational_02.py

Do you understand the problem with this?

```
print(1 == 2)
print(1 = 2)
```

120.7 Functions 2 - indentation and your own functions [func-01-simple.rst]

In Python, indentation of text from the left is significant. It is used to group and indicate blocks of code. By convention 4 spaces are used.

120.7.1 Note

- If you are using a simple text editor, change the tab-key configuration to indent as 4 spaces.
- Although **tabs** (tab characters) can be used instead of spaces, it is still better to use spaces.
- Either way, be consistent within a file.

120.7.2 first_function.py

Write and test the following program, `first_function.py`.

```
def first_function():
    print("Hello, from first_function.")
    print(5 * 4)
    print("Bye, from first_function.")

print("Outside function.")
first_function()
first_function()
print("End of program.")
```

120.7.3 ott_comments_first_function.py

Copy the previous code, `first_function.py` , into a new program `ott_comments_first_function.py` . Write a comment above each line explaining what it does.

120.7.4 three_times_first_function.py

Copy the `first_function.py` code into a new program, `three_times_first_function.py`. Instead of two, call the function three times. Test and write a comment above each line explaining what is going on.

120.7.5 my_function_twice.py

Write a program, `my_function_twice.py`, containing a function, `my_function()`.

This function should print a friendly message. In your program, call this function twice.

120.7.6 doubles.py

Create a program, `doubles.py`, with the following specification:

- It should contain one function.
- It should display the result of `2 * "double "`.
- Call your function 3 times.

The output should be:

```
double double
double double
double double
```

120.8 Errors - pass [func-02-pass.rst]

`pass` is a place holder statement that does nothing other than tell Python that you meant to do nothing. It can be useful when you are developing a program.

120.8.1 stubs_error.py

Write and test this program, `stubs_error.py`, and note the output.

```
def func_1():

def func_2():

def func_3():

func_1()
func_2()
func_3()
```

120.8.2 `stubs_fixed.py`

Using `pass` fix the previous program and save as `stubs_fixed.py`. Check that it now runs without an error.

```
def func_1():
    pass

def func_2():
    pass

def func_3():
    pass

func_1()
func_2()
func_3()
```

In this way you can use *function stubs* to help plan your code.

120.9 `if` 1 - introduction [if-02-if.rst]

The `if` statement is used to check a condition and selectively execute a block of code. The `if` block is indicated by indentation (recommended 4 spaces).

120.9.1 `if_01.py`

Write and test this program.

```
left_var = 15
right_var = 3 * 5

if left_var == right_var:
    print("They are the same value.")
    print("Inside the 'if' block.")

print("End of program.")
```

120.9.2 `if_02.py`

Make a copy of `if_01.py` and save as `if_02.py` .

- Change the == into a `!=`.
- Make the `print` messages appropriate.
- Change the `5` to `"5"`.
- Comment your code to demonstrate that you understand what is happening.

120.10 if 2 - if, else [if-04-else.rst]

In this section the `if...else` statement is introduced. When the `if` condition is false the `else` block is executed.

120.10.1 if_else.py

Write and test this program.

```
var = 'a'
if var == 'a':
    print("We have an 'a'.")
else:
    print("We don't have an 'a'.")

print("That's the end of the 'a' test.")
```

120.10.2 if_else_not_a.py

Saving as `if_else_not_a.py`, copy the previous program, change the variable value from `a` to another value and test. Finally, explain your code with comments.

120.11 Functions - return [func-03-return.rst]

120.11.1 second_function.py

Write and test the following program, `second_function.py`.

```
def second_function():
    val = 4 + 5
    return val

returned_value = second_function()
print(returned_value)
```

120.11.2 second_func_ott_comments.py

Write a well commented version of **second_function.py** and save it as **second_func_ott_comments.py**.

120.11.3 doubles_55.py

Write a program, **doubles.py**, that doubles 55 and returns the result. Display the returned result.

120.12 Operators - modulo (%) operator [op-03-modulo.rst]

The **modulo** (%) operator gives the **remainder** of a division of the left hand side by the right hand side of the %.

120.12.1 modulus.py

Write a commented program, modulus.py, to carry out the following calculations. You will want to experiment in interactive mode.

```
print(1 % 3)
print(90 - 25 * 3 % 4)
print(4 + 3 + 2 - 6 + 5 % 3 - 2 / 5 + 7)
```

120.12.2 modulus2.py

Predict before checking the results of these sums.

```
print(123 % 2)
print(1234 % 2)
```

120.13 Functions - parameters [func-05-param.rst]

As well as providing return values, functions can also take inputs.

120.13.1 double.py

What is the output from these two calls to double?

```
def double(var):
    return 2 * var

print("double(2) =", double(2))
print("double('python ') =", double('python '))
```

120.13.2 even.py

Write and call a function is_even that that takes one argument and returns True if the input value is even otherwise False. Hint % operator.

120.13.3 multiply_three.py and iterative development

It is a good idea to build these up in stages, testing as you go. Write and test the following program, multiply_three.py. Write small sections and test as you go along as follows.

120.13.4 multiply_three.py

Write and test. If you do not already know, ask the internet what pass does.

```
def multiply_three_parameters():
    pass

print(multiply_three_parameters())
```

120.13.5 multiply_three.py

Write and test.

```
def multiply_three_parameters(a):
    return a

print(multiply_three_parameters(1))
```

120.13.6 multiply_three.py

Continue your writing and testing until you have completed this program.

```
def multiply_three_parameters(a, b, c):
    x = 2 * a
    y = 2 * b
    z = 2 * c
    return x, y, z

print(multiply_three_parameters(1, 2, 3))
print(multiply_three_parameters(1.0, 2.2, 3.3))
print(multiply_three_parameters('hello', 2.2, 3.3))
d, e, f = multiply_three_parameters('hello', 2.2, 'bye')
print(d, e, f)
```

120.13.7 multiply_three_ott_comments.py

Copy the previous code, multiply_three.py, into a new program multiply_three_ott_comments.py. Write a comment above each line explaining what it does.

120.13.8 fahrenheit_to_celsius.py

Write a program, fahrenheit_to_celsius.py, that contains a function to convert Fahrenheit to Celsius.

120.13.9 celsius_to_fahrenheit.py

Write a program, celsius_to_fahrenheit.py, that contains a function to convert Celsius to Fahrenheit.

120.14 input 1 - introduction [io-01-input.rst]

120.14.1 input_name.py

Write and test the following program, input_info.py.

```
print("What is your first name?")
name = input()

print("Hello ", name)
```

120.14.2 input_age.py

Write a similar program to input_name.py that asks for and reads in a user's age, input_age.py .

120.15 Files 1 - introduction [file-01-read.rst]

Like most languages, Python can read data from files.

120.15.1 line_numbers.txt

With a text editor, create the file line_numbers.txt, with the following content:

```
Line one
Line two
Line three
Line four
Line five
Line six
```

120.15.2 file_reader.py

Write a program file_reader.py to read the text from the line_numbers.txt file you created. The r mode is to tell your operating system that you are opening the file for **r**eading.

```
file_name = "line_numbers.txt"

mode = 'r'
f = open(file_name, mode)

text = f.read()
f.close()

print(text)
```

120.15.3 file_high_read.py

Using the following text file, high.txt:

```
Python is a high-level
programming language.
A program is a sequence
of instructions.
```

- Create a program, file_high_reader.py, to read in the file high.txt and display the contents to the console. Make sure you close the file once you have finished with it.

120.15.4 file_readline.py

- Write this program to read in one line of high.txt and display it to the console.

  ```
  f = open('high.txt')
  line = f.readline()
  print(line)
  f.close()
  ```

120.16 Indexing 1 [index-01.rst]

120.16.1 Strings 4 - indexing

Characters in strings are accessed using square brackets. The first element is at the index zero, e.g. a_string[0].

120.16.2 alphabet_hello.py

Write and test the following code, alphabet_hello.py.

```
a_str = "abcdefghijklmnopqrstuvwxyz"
print(a_str[7] + a_str[4] + a_str[11] + a_str[11] + a_str[14])
```

120.16.3 alphabet_my_name.py

Write a program alphabet_my_name.py. This should use the same a_str and indexing technique as alphabet_hello.py to display your name.

120.16.4 Lists 1 - lists introduction and indexing

A Python **list** is a container for things (a grouping of objects) in an organised order. A list is created using square, brackets i.e. []. You will use them often.

120.16.5 empty.py

Create an empty list.

```
an_empty_list = []
print(an_empty_list)
```

120.16.6 ten.py

Create and access a list of 10 integers 0 to 9. Write and test ten.py.

```
ten_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(ten_list)
print(ten_list[0])
print(ten_list[6])
```

120.16.7 ten_ott_comms.py

Explain ten.py with comments in ten_ott_comms.py.

120.16.8 Lists 2 - mutable

You can access and change items in a list. (Note, you can not change letters of a string in place.)

120.16.9 list_three.py

Create and run this program.

```
a_list = ["one", "two", "three"]

print(a_list)
print(a_list[1])
print(a_list[1][0])

a_list[1] = 2
print(a_list)
```

120.16.10 list_three_ott_comms.py

Explain `list_three.py` with comments in `list_three_ott_comms.py`.

120.16.11 Tuples 1

A `tuple` is another Python container. Like `lists` you access members by position, however, you can not modify the members of a **tuple**. A **tuple** is created by separating data with commas separating entries. A **tuple** is often delimited by brackets i.e. `()`.

120.16.12 tuple_simple.py

Try this program.

```
t1 = (1, 2, 3)
t2 = (4, 5, 6, 7)
t3 = t1 + t2
print(t1, t2, t3)
```

120.16.13 Dictionaries 1

A Python **dictionary**, similar to a **list**, is a container for objects. Unlike lists, which you access by position, dictionary members are accessed by **key** values. A dictionary is delimited with curly brackets i.e. **{}**.

120.16.14 french_01.py

Write, test and comment the following program, `french_01.py`.

```
french = {'yes':'oui', 'no':'non', 'hello':'bonjour'}
print(french['hello'])
```

120.16.15 `french_02.py`

Write a program `french_02.py` that uses the dictionary

`french = {'yes':'oui', 'no':'non', 'hello':'bonjour'}`

to display `ouiouioui`.

120.16.16 `french_03.py`

Write, test and explain in comments this sweet program,`french_03.py`.

```
french = {'yes':'oui', 'no':'non', 'hello':'bonjour'}
print(french['hello'][3]
      + french['yes'][1]
      + french['no'][2]
      + french['yes'][2]
      + french['no'][1]
      + french['hello'][-1])
```

120.17 Looping 1 - a simple `while` loop [loop-01-while.rst]

A `while` loop will keep executing a code *block* of code as long as the boolean expression controlling it is `True`. Although useful, `while` loops are not used in Python as much as other languages.

120.17.1 `simple_while_loop.py`

Write, test and comment each line of this code, `simple_while_loop.py`.

```
x = 0
while x < 10:
    x = x + 1
    print(x)
```

120.17.2 `while_two_step.py`

Copy the previous `simple_while_loop.py` program and save as `while_two_step.py`. Make modifications so that the output is:

```
2
4
6
8
10
```

120.18 Looping 2 - for loop [loop-02-for.rst]

In Python, a for loop, loops over the items in any sequence. The `for ... in ... :` statement manages the process and stops when there are no items left. for loops are extremely useful in Python and you will use them often.

120.18.1 simple_for.py

Write and test the the following program, `simple_for.py`.

```
list_of_names = ["Jim", "Julia", "John"]

for name in list_of_names:
    print("Hello", name)
```

120.18.2 int_for.py

Fix, test and comment the following `int_for.py` program. Hint: `[1, 2, 3]`.

```
for item
    print(item + 1)
```

The output should be:

```
2
3
4
```

120.19 Modules 1 - import [module-01-import.rst]

In Python, a **module** is a file that contains code that can be used in your program. For example, the `math` module provides some useful mathematical functions. To bring this into your program use the `import` Python keyword.

120.19.1 import math

Try this in your Python interactive shell (>>>).

```
> python

>>> import math

>>> help(math)

This module is always available. It provides access to the...

>>> math.pi
```

120.19.2 trig.py

Write and test the program `trig.py`.

```
import math
print(math.sin(math.pi/2))
```

120.19.3 trig_comments.py

Copy `trig.py`, explain what is going on with code comments and save as `trig_comments.py`.

120.19.4 trig_var.py

Using a variable for `math.pi/2`, rewrite `math.sin(math.pi/2)` over two lines, printing our your results and save as `trig_lines.py`.

120.20 Errors 2 - try and except [except-01-try.rst]

Errors detected during program execution are called *exceptions*. Exceptions can be triggered automatically on errors, or manually in your code. They can be intercepted and acted upon by your code and allow your program to continue. However, especially during development, on an error you will want the program to stop. In short, exceptions can be used, especially in an **emergency**, to abandon all the current activity and allow for jumps out of large chunks of code.

Given an exception, `try/except` can be used to handle and allow your code to continue. The general `try/except` syntax is:

```
try:
    pass
    # primary action
except:
    pass
    # run if any exception raised
```

Practice error handling with the following exercise.

120.20.1 broken.py

This is a broken piece of code. Write, run and make a note of the error message.

```
print(int(2.1))
print(int('2'))
print(int('cat'))
```

120.20.2 handled.py

Here, instead of fixing, in this version of the previous code the error is handled. Write, test and experiment with this code, `handled.py`.

```
print(int(2.1))
print(int('2'))

try:
    print(int('cat'))
except ValueError:
    print('cat is not a number')
```

120.21 Functions 3.1, Scope 1 and Variables - introduction to *scope* [func-08-scope.rst]

Scope concerns the *visibility* of names in your code. The *scope* of a variable in a program is the lines of code in the program where the variable can be accessed. For example, a variable name with *global* scope can be seen and accessed anywhere in a file. Generally, avoid using global variables but if you must then try not to change them. A variable defined in a function is *local* to that function. Scope is better understood with practice.

120.21.1 scope_01.py

Predict the output of the following code snippet and test your answer.

```
x = 1 # Global (file) scope.

def scope_function():
    x = 2 # Local (function) scope.
    print(x)
print(x)
scope_function()
print(x)
```

120.21.2 scope_02.py

Predict the output of the following code snippet and test your answer.

```
x = 1 # Global (file) scope.

def scope_function():
    print(x)
print(x)
scope_function()
print(x)
```

120.22 Indexing - slicing [index-05-ctd.rst]

120.22.1 alphabet_slicing.py

To create sub-strings you can use string *slicing*. Comment and explain this code, `alphabet_slicing.py`.

```
alphabet_string = "abcdefghijklmnopqrstuvwxyz"
print(alphabet_string[0:5])
print(alphabet_string[:5])
print(alphabet_string[1:5])
print(alphabet_string[-25:5])
print(alphabet_string[-2:])
print(alphabet_string[24:])
print(alphabet_string[24:26])
print(alphabet_string[0:13:2])
```

120.22.2 alphabet_slicing_2.py

Using slicing of `alphabet_string = "abcdefghijklmnopqrstuvwxyz"` give the following outputs:

```
bcdefghijklmnopqrstuvwxyz
adgjmpsvy
ghijklmnopqr
```

120.22.3 glasgow.py

Using the string `Glasgow` and only two string slices, create the new string `gowGla`.

120.22.4 Lists 4 - slicing list_slicing.py

Write, test and comment this program, `list_slicing.py`, which is an example of lists and *slices*. Try to predict the output before you run your code.

```
number_list = [1, 2, 3, 4, 5]
print(number_list[1:3])
print(number_list[0:3])
print(number_list[0:5:2])
print(number_list[1:])
print(number_list[-2:5])
```

120.22.5 Lists 4 - slicing `two_three_four_slice.py`

Write your own slicing program, using `number_list = [1, 2, 3, 4, 5]` and the number `-1` and `1` in a way to output `[2, 3, 4]`. Save your program as `two_three_four_slice.py`.

120.23 Indexing - list methods [index-06-ctd.rst]

An object, such as a list, can have internal functions called methods. They are accessed using *dot* notation.

120.23.1 `list_methods.py`

Write and test this program, `list_methods.py`.

```
list_of_numbers = [1, 2, 3, 4]
list_of_numbers.append(1)
list_of_numbers.pop()
list_of_numbers.extend([11, 12, 13])
list_of_numbers.pop(2)
list_of_numbers.reverse()
list_of_numbers.insert(3, 21)
list_of_numbers.sort()
print(list_of_numbers)
```

120.23.2 `list_methods_ott_comments.py`

Make another copy of `list_methods.py`, comment each line and save as `list_methods_ott_comments.py`. Use `print` to understand what each line of this program does.

120.23.3 `list_complex.py`

Nested lists and append method. Write and test `list_complex.py`. You may be surprised to find that this program outputs `Glasgow is hot!`.

```
nested_list = []
nested_list.append('glasgow')
nested_list = nested_list + [ 1 , 1.0, [3, 5], 7]
nested_list.append(['is', 'wet'])
nested_list[5][1] = 'hot'
print(nested_list[0].capitalize(), nested_list[5][0], nested_list[5][1] + "!")
```

120.23.4 list_complex_ott_cmts.py

Copy list_complex.py, carefully comment and save as list_complex_ott_cmts.py.

120.23.5 list_complex2.py

Write a program, list_complex2.py, that uses nested_list = [1, [200, 300], 4] to write 1 200.

120.23.6 list_range.py

List and range. Predict and test the result of this code?

```
numbers = range(1, 10, 2)
print("numbers[1:] = ", list(numbers[1:]))
```

120.23.7 lists_7.py

Lists len and modulo. Given the two lists, below, which of the following 4 code snippets give the result 7? Try to figure out the answer before testing.

```
list_1 = [0, 1, 2, 3, 4, 5, 6, 7]
list_2 = [1, 2, 3, 4, 5, 6, 7]

print(len(list_1))
print(len(list_2))
print(len(list_1) % len(list_2))
print(len(list_2) % len(list_1))
```

120.23.8 out_of_sorts.py

Lists and sort. What do think the output of this code will be? Try to gess the answer before running the code.

```
out_of_sorts = [1,3,2,4,3,5,4,6,5,7]
out_of_sorts.sort()
print("out_of_sorts.sort()", out_of_sorts)
```

120.23.9 list_reverse_method.py

List reverse method. Write a program that uses a list method to change [1, 2, 3, 4, 5, 6] to [6, 5, 4, 3, 2, 1]. Display the start list and result.

120.24 Classes 1 - A first look at *classes* [class-01-methods.rst]

- Like functions, **classes** are a way of grouping and organising your code.
- Classes are associated with a programming style called *object oriented* programming.
- You will find out more about classes and there use in the following exercises.

120.24.1 first_class.py

Write and test the following code, `first_class.py`.

```
class FirstClass:
    def method(self):
        print("Hello, from the class.")

instance_of_class = FirstClass()
instance_of_class.method()
```

120.24.2 first_class_ott_comments.py

Make a copy of `first_class.py` code and save it as `first_class_ott_comments.py` and add a comment to each line explaining what it does.

120.24.3 class_new_method.py

Make a copy of `first_class.py` code and save it as `class_new_method.py`. Add and test your own new method.

120.25 Waypoint [preamble-10-waypoint.rst]

Well done at getting this far through the work. Already, you know enough Python to put it to some practical use. In the following sections you will reinforce what you have learned, extend your knowledge and learn some new concepts. There will be repetition, however, do not skip any sections and go at a steady pace. Even if revision, it will still be excellent practice.

CHAPTER

ONE

EXERCISES 2

121.1 Environment - IDLE IDE [preamble-02.rst]

IDLE is a basic **IDE** (integrated develop environment) which *usually* comes as part of a default Python installation. IDLE is intended to be simple and suitable for beginners in an educational environment. However, do not expect too much from it and **do not** use it for GUI development.

121.1.1 Launch IDLE

Launch IDLE (we may need to discuss, or ask the Internet, how best to do this).

```
> idle3
```

121.1.2 Use the interactive shell

In the interactive **shell** (indicated by 3 chevrons, >>>) say something nice e.g.

```
>>> print("Hi")
```

121.1.3 Create a program from IDLE, `idle_01.py`

Select **File > New Window**.

In the new window (editor) select **File > Save as...** then `yourname_python_code/idle_01.py`.

In the editor write the following code.

```
message = "This was created using IDLE.\n"
print(10 * message)
```

Run the program, in the editor **Run > Run Module**.

121.2 Strings - embedding quotes [string-02.rst]

Strings are a series of characters contained in quotes. Double or single quotes are equivalent and can be used in combination to embed in the other. (Alternatively, to embed quotation marks inside strings you can use the escape \ character.)

121.2.1 burns.py

Demonstrate that you understand the embedding and escaping of quotes by writing and commenting each line of burns.py.

```
print("O my Luve's like a red, red rose,")
print("That's newly sprung in June:")
print('O my Luve\'s like the melodie,')
print('That\'s sweetly play\'d in tune.')
print(5 * "-")
print('By "Rabbie" Burns')
```

121.3 Logic [if-03-logic_relational.rst]

121.3.1 logic_a_b.py

Given that A = 5 and B = 6, predict the output before running the following:

```
A = 5
B = 6
print(not(A > B) and (A < B))
print(not(True) or (A > B))
```

121.3.2 logic_1.py

Try to figure out the answer to each question before you run the code.

Given that **a = 1** and **b = 2**, are the following print outputs **True** or **False**?

```
a = 1
b = 2

# 1
print(a > b)

c = a + a

# 2
print(c > b)

c = a + a

# 3
```

(continues on next page)

(continued from previous page)

```
print(c <= b)

# 4
print(b != a)

# 5
c = b
d = a + a

# 6
print(c != d)

d = a + a + a
c = a + a + b

# 7
print(d == c)

c = a - b
d = b - c

# 8
print((b + d) != (a + c))

c = a
d = c - a

# 9
print(d <= c)

c = 3
d = 2

# 10
print((a + c) < (b + d))

c = 12
d = a + b
c = c + d

# 11
print(c >= (d + 4))
```

121.3.3 Logic - logic and relational operators `logic_and_relations.py`

Before running, predict the output of this program, `logic_and_relations.py`.

```
print(True)
print(False and False)
print(False and True)
print(True and False)
print(True and True)
print(True or 1 == 2)
print(not (True and True))
print(2+1 == 3 and not("left" == "right" or "good" != "fun"))
print(1 + 1 == 2 and 1 + 1 != 3)
print(1 + 1 == 2 and 1 + 1 == 3)
```

121.4 Operators - basic operators, numbers and maths [numbers-01.rst]

121.4.1 `counting_sheep.py`

Write the following `counting_sheep.py` program, designed to help you sleep, then run the program in your terminal.

```
print("Number of sheep:")
print("Ewes", 30 + 30 / 6)
print("Rams", 90 - 25 * 3)
print("Count the lambs:")
print(4 + 3 + 2 - 6 + 5 * 3 - 2 / 5 + 7)

print("Is 4 + 3 less than 6 - 8?")
print(4 + 3 <  6 - 8)
print("Add 4 and 3?", 4 + 3)
print("Subtract 6 from 8?", 6 - 8)
print("That explains why it is False.")

print("Some more.")
print("Greater?", 6 > -3)
print("Greater or equal?", 6 >= -3)
print("Less or equal?", 6 <= -3)
```

121.4.2 `counting_sheep_ott_comments.py`

Make sure you understand what is going on with the previous program. The order in which the operators are applied can impact on the answer.

- Make a copy of `counting_sheep.py` and save as `counting_sheep_ott_comments.py`.
- Above each line of `counting_sheep_ott_comments.py`, use the # to write a comment explaining to yourself what the line does.

121.4.3 counting_sheep_floating.py

- Rewrite `counting_sheep.py` as `counting_sheep_floating.py` to use floating point numbers (hint: 10.0 is floating point).

121.4.4 float_10.py

What do you think the result of this code snippet will be?

```
print("float('10')", float('10'))
```

121.4.5 int_10.py

What do you think the result of this code snippet will be?

```
print(int(-10.9))
```

121.4.6 div_15_2.py

What is the output of the following?

```
print("15/2 = ", 15/2)
```

121.4.7 kmph_to_mph.py

Given that there are about 1.61 kilometres in a mile, if you jog 5 kilometres in 20 minutes and 22 seconds, what is your average speed in miles per hour? Call your program `kmph_to_mph.py`.(9.15 mph)

121.4.8 mileage.py

Write a short program to calculate the amount someone would expect for a work mileage rate of 45p for the first 10,000 miles and 20p thereafter in a year?

121.4.9 three_point_5.py

Which of the following give **3.5**? There are more than one.

```
print(7//2.0)
print(7/2)
print(7.0//2)
print(7.0/2.0)
```

121.4.10 x1_and_x2.py

Before running, predict what the output of this code will be?

```
x1 = 100
x2 = 3
x1 = x1 + 10
x1 = x1 + x2
x2 = x1
x2 = x2 + x1
print(x2)
```

121.4.11 remainder.py

What is the result of this **floor** division?

```
print("3.0//2.0 = ", 3.0//2.0)
```

121.4.12 remainder7.py

Which of the following give **7.0** as a result? There are more than one.

```
print(15//2.0)
print(15/2)
print(14/2)
print(15//2)
print(15.0//2)
```

121.5 Variables 3 - variables and ``*type*`` conversions [var-03.rst]

There are times when you will need to convert between data types. You can achieve this with built in functions like **int(), long() and float()**. Use these to convert *strings* input by a user into a numbers. However, you should take care as information can be lost.

121.5.1 one_int_type.py

This program uses `type` and `int` *built-in* functions. Write this `one_int_type.py` program and try to guess the output before you run the code.

```
a = 1
b = '1'
c = 1.0
print(a, b, c)
print(type(a))
print(type(b))
print(type(c))

d = int(b)
```

(continues on next page)

(continued from previous page)

```
e = int(c)

print(d, type(d))
print(e, type(e))
```

121.5.2 one_int_type_ott_comments.py

Saving as one_int_type_ott_comments.py, copy and comment the previous one_int_type.py.

121.5.3 five_type.py

Create a program five_type.py that:

- Allocates the integer 5 to variable name five_int.
- Allocates the floating point number 5 to the variable name five_float.
- Allocates the string 5 to the variable name five_string.
- By printing the type, proves that you have allocated the the types correctly.
- Using the built in function float, multiply five_int, five_float and five_string to give 125.0.

121.5.4 ten_convert.py

Use your knowledge of conversions to fix the program **ten_bad.py**. Save your program as **ten_convert.py**.

```
# ten_bad.py

print(10, '10', 10.0)
print(type(10))
print(type('10'))
print(type(10.0))
print(10/10.0, type(10/10.0))
print(10 * '10', type(10 * '10'))
print(10 / '10', type(10 / '10'))
```

121.5.5 ten_convert_ott_comments.py

Copy ten_convert.py fully comment and save as ten_convert_ott_comments.py.

121.6 Classes - `return` from methods [class-02-return.rst]

Similar to functions, an objects method can return values.

121.6.1 `method_return.py`

Write this program, test, experiment and explain in comments what is going on.

```
class ClassWithReturnMethod:

    def get_double_hi(self):
        return 2 * "Hello "

instance_of_class = ClassWithReturnMethod()
val = instance_of_class.get_double_hi()
print(val)
```

121.7 Help 2 [help-03-dir.rst]

121.7.1 *dir*

Typically, used during an *interactive session*, `dir` is another useful way of finding help during your Python session. It will tell you what names are in your *local scope* or *attributes* for objects. Try the following.

- First, launch Python in interactive mode.

  `> python`

- Then, in interactive mode…

  `>>> dir() # names in local scope`

  `>>> dir(1.1) # attributes of a float`

  `>>> dir("hi") # attributes of a string`

  `>>> help(dir) # more information on dir`

121.8 Loops [loop-03.rst]

121.8.1 `while_3.py`

What do you think the output of this code fragment will be? Test your answer.

```
x = 0
while x < 3:
    print(x)
    x += 1
```

121.8.2 while_123.py

Use a `while` loop over `three_list = ["one", "two", "three"]` to display each string to the console.

121.8.3 types_in_list.py

Write a short program that uses a `for` loop to display the type of each member of the list `[123.4, "one two three four", 1234]`.

121.9 input - inputting data and % string formatting expression [io-02-input.rst]

The `%` operator is used to created formatted strings.

121.9.1 input_info_1.py

Write and test the following program, `input_info_1.py`.

```
name = input("What is your first name? ")
surname = input("What is your surname? ")
age = input("How old are you? ")

print("Hello %s %s, you are %s years old." % (name, surname, age))
```

121.9.2 input_info_2.py

Write a program `input_info_2.py` to ask a user a question.

121.10 Modules 2 - import with from [module-02.rst]

Below we experiment with different ways of using `imports` to bring outside code into your program.

121.10.1 mixed.py

This program, `mixed.py`, uses *inbuilt* (no `import`) functions, the `math` module (`import math`) and the `random` module (`import random`). Write, test and comment each line. You may need help from the Internet.

```
import math
import random

print(math.e)
print(abs(-1.1))
print(math.floor(1.5))
print(math.ceil(1.5))
print(round(1.23456))
print(round(1.23456, 3))
```

(continues on next page)

(continued from previous page)

```
print(sum([1, 2, 3]))
print(math.trunc(12345.67))
print(pow(3,2))
print(3**2)
print(3.0**2)
print(math.pow(3,2))
print(math.sqrt(4))
print(divmod(7, 2))
print(random.randint(0, 9))
print(random.randint(0, 9))
```

121.10.2 from

import using from…

Write and test these two programs, import_math.py and import_pi.py, alternative methods for importing code.

121.10.3 import_math.py

```
import math
print(math.pi)
```

121.10.4 import_pi.py

```
from math import pi
print(pi)
```

121.11 Tkinter module [projects-01.rst]

Tkinter is a basic and limited GUI toolkit but has the advantage that it is usually installed at the same time as a standard Python installation.There are many powerful GUI tool kits that can be used with Python. If you are interested then search the internet for *python gui toolkits*.

121.11.1 hello_gui.py

Write and test the following code, hello_gui.py. I recommend that you run this from the terminal and not through a development environment. It is possible that an **IDE** (check your documentation) may encounter *threading* problems.

```
from sys import exit
from tkinter import *
root = Tk()
Button(root, text='Hello World!', command=exit).pack()
root.mainloop()
```

121.12 if 3 - if, elif and else [if-05-elif.rst]

Within an if / else clause you can insert multiple elif statements.

121.12.1 travel_info.py

Write and test this program, save as control/travel_info.py.

```
people = 30
train_seats = 40
if train_seats > people:
    print("Take the train.")
elif train_seats < people:
    print("Do not use the train.")
else:
    print("Not sure.")
```

121.12.2 ott_comments_travel_info.py

Write a fully (over) commented version of travel_info.py and save as ott_comments_travel_info.py.

121.12.3 travel_info2.py

When there are multiple code branching being controlled by elifs, it can be confusing. Try to keep your code simple and easy to understand. The following is not good, clear code.

Make a copy of travel_info.py and save as travel_info2.py.

- Add another elif statement which advises to take the train if you are running late.
- Change the number of people.
- Add a boolean for running late.
- Explain what is going on.

```
people = 40
train_seats = 40
running_late = True
if train_seats > people:
    print("Take the train.")
elif train_seats < people:
    print("Do not use the train.")
elif running_late:
    print("Just take the train. You are late.")
else:
    print("Not sure.")
```

121.13 `input` 3 and Conversion 2 - conversions and `input` [type-01.rst]

121.13.1 `convert_broken.py`

Write and test this `convert_broken.py`. It should fail with an error message like

> `TypeError: cannot concatenate 'str' and 'int' objects`

```
print("What is your first name?",)
name = input()
surname = input("What is your surname?",)
print("How old are you?",)
age = age + 1
print("In one year %s %s will be %d years old." % (name, surname, age))
```

121.13.2 `convert_fixed.py`

Copy your previous broken code `convert_broken.py` and, saving as `convert_fixed.py` , make a change so that it now works. (Hint `age = int(age_str)`)

121.13.3 `convert_ott_comments.py`

Copy the working code `convert_fixed.py` into `convert_ott_comments.py`. With comments, clearly explain why this works.

121.14 Looping 2 [loop-04.rst]

In this exercise you are going to write two programs. They will both achieve the same outcome but one uses a `while` loop whereas the other uses a `for` loop.

The out put of both programs should be:

> `PPyytthhoonn`

121.14.1 `expand_string_while.py`

First, write test and comment this program.

```
string_in = "Python"
string_out = ""

character_position = 0
while character_position < len(string_in):
    char = string_in[character_position]
    string_out = string_out + 2 * char
    character_position = character_position + 1

print(string_out)
```

121.14.2 expand_string_for.py

Now, write, test, comment and compare this version of the program. Every time through the `for` loop, the next character in the string is assigned to the variable char. The `for ... in ... :` statement manages the process and stops when there are no characters left.

```
string_out = ""
for char in string_in:
   string_out = string_out + 2 * char

print(string_out)
```

121.15 Loops - continue [loop-05.rst]

The `continue` statement allows the program flow to jump to the top of the containing loop.

121.15.1 continue_loops.py

Write and test this program, `continue_loops.py`. Add some judicious comments to explain what is going on.

```
alphabet_string = "abcdefghijklmnopqrstuvwxyz"

index = 0
while(index < len(alphabet_string) - 1):

    index = index + 1
    if (index % 2 == 0):
        continue
    print(alphabet_string[index])


count = 0
for letter in alphabet_string:

    count = count + 1
    if(count % 2 == 0):
        continue
    print(letter)
```

121.16 Looping - break [loop-10.rst]

The keyword `break` is used to immediately exit the containing loop.

121.16.1 break_loops.py

Write and test the following break_loops.py.

```
# break_loops.py

print("Loop 1")
num = 0
while(True):
    print(num)
    num = num + 1
    if num == 6:
        break

print("\nLoop 2")
for num in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    if num == 6:
        break
    print(num)
```

121.16.2 break_while.py

On the input of q, use a break statement to exit the following programs while loop.

```
# break_while.py

while(True):
    user_in = input()
    if user_in == 'q':
        break
```

121.16.3 range_10.py

List and range…

What is the output of this code?

```
# range_10.py

range_10 = range(10)
print("range_10 = range(10), range_10[3:] = ", list(range_10[3:]))
```

121.16.4 range_3_10.py

for and range…

Before running, predict the output of the following code?

```
# range_3_10.py

for i in range(3, 10):
    print(i)
```

121.16.5 odd_even.py

for, lists and modulus

Write a short program that uses a **for** loop, **range(100)** and the modulus operator (%) to display the numbers between 0 and 99 with a short message indicating if each is **even** or **odd**.

121.16.6 zipping.py

Lists - *`for``*, *``list``* and *``zip``*,

Investigate the use of *zip* to figure out what the following code will do, zipping.py?

```
nine_list = [1,2,3,4,5,6,7,8,9]
nineteen_list = [11,12,13,14,15,16,17,18,19]
print("zip(nine_list, nineteen_list) = ", zip(nine_list, nineteen_list))

answer = ""
for i, j in zip(nine_list, nineteen_list):
    answer = answer + str(i) + " " + str(j) + " "
print(answer)
```

121.17 Functions - Recursion [func-04-recursion.rst]

A **recursive** function is a function that calls itself from within itself. The process is called **recursion**.

121.17.1 recursive_function.py

Write and test this recursive function.

```
def recursive_function(n):

    print(n)
    if n >= 1: #recursive condition
        n = n - 1
        return recursive_function(n)
    else:
        return(n) #end
```

(continues on next page)

(continued from previous page)

```
print(recursive_function(-1))
print(recursive_function(4))
print(recursive_function(4.1))
```

121.17.2 recursive_function_ott_comments.py

Make a copy of the previous code and thoroughly comment.

121.18 Errors 3 - error handling and input [except-02-input.rst]

121.18.1 error_handling.py

Using `input()` could crash if you get unexpected input. Write and test this program, `error_handling.py`. Explain the key word `continue` and error handling `try/except` code works in your comments.

```
while(True):

    user_input = input(
        "Input an integer number 1 to 10 ('q' to quit): ")

    if user_input == 'q':
        break

    try:
        user_input = int(user_input)
    except:
        print("Boo! That is not an integer.")
        continue

    if (1 <= user_input <= 10):
        print("Great! That's a valid number.")
    else:
        print(
            "Boo! That integer is not in the range 1 to 10.")
```

121.19 Errors - raise [except-03-raise.rst]

As well as Python giving errors automatically, you can also signal errors using `raise`.

Write and test the following programs and explain in comments what is going on.

121.19.1 raise_01.py

```
while True:
    try:
        x = int(input("Give me a number: "))
        break
    except ValueError:
        print("Try again.")
```

121.20 Files 2 - writing 1 [file-03-write.rst]

We have previously looked at *reading* files. To do the opposite and *write* to a file, you have to open it in write mode. This is done by setting the second parameter of the function `open` to `'w'` write mode - `open('file_name','w')`.

121.20.1 file_writer.py

Write and test this program, `file_writer.py`. It saves a message to a file `text.txt`.

```
file_name = "text.txt"
mode = 'w'
f = open(file_name, mode)

text = "Hi, from the test file."
f.write(text)

f.close()
```

121.20.2 Check

After running the program, check it worked by opening `text.txt` with your editor.

```
Hi, from the test file.
```

121.20.3 writelines

The file `writelines` method writes all the strings in a list into a file.

121.20.4 file_writelist.py

Write and comment this program `file_writelist.py` that uses `writelines` to save the entries of a *list* to a file.

```
num_list = ['0\n','1\n','2\n','3\n','4\n','5\n','6\n','7\n','8\n','9\n']

file_name = "data.dat"
mode = 'w'

f = open(file_name, mode)
```

(continues on next page)

(continued from previous page)

```
f.writelines(num_list)

f.close()
```

121.20.5 data.dat

Use your editor to open the file you created, `data.dat`, and check your previous program worked correctly.

```
0
1
2
3
4
5
6
7
8
9
```

121.20.6 file_readlist.py

Write a program `file_readlist.py` to read your list file and compare it against the original.

121.20.7 file_writelist_2.py

Write a program, `file_writelist_2.py`, that saves a list. Check that the file `data_2.dat` was created correctly. Write a program to read in the previous **data.dat** into a list of integers.

```
file_name = "data_2.dat"
mode = 'w'
f = open(file_name,mode)

num_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

for item in num_list:
    f.write(str(item) + '\n')

f.close()
```

121.20.8 file_hello_writer.py

Write a program, `file_hello_writer.py`, that writes a file, `hello.txt`, containing the message, `Hi, from the test file.`.

```
file_name = 'hello.txt'
mode = 'w'
f = open(file_name, mode)

text = "Hi, from the test file."
f.write(text)

f.close()
```

121.20.9 file_sat_read_write.py

Create a program, `file_sat_read_write.py`, to write text from a file `saturday_in.txt` (below) into the file `saturday_out.txt`. Make sure you close the file once you have finished with it.

121.20.10 saturday_in.txt

```
Well at least
it is
Saturday.
```

121.20.11 file_sat_read_write.py

```
f_in = open('saturday_in.txt')
contents = f_in.read()
f_in.close()

f_out = open('saturday_out.txt', 'w')
f_out.write(contents)
f_out.close()
```

121.21 Modules [module-03.rst]

More `importing`.

121.21.1 single_variable.py

Modules and your own code `import`. Write your own module module `single_variable.py`.

```
variable = 100
```

121.21.2 main.py

Write the program `main.py` that utilises your `single_variable.py`.

```
import single_variable

print(single_variable.variable)
```

121.21.3 import variable

Given a python script with only one line, `variable = 1`, which of the following gives an output on the interactive prompt of 1?

- A

```
import script_2
print(script_2.variable)
```

- B

```
import script_2
print(variable)
```

121.22 Comments 2 - triple quotes [help-04-triple_quotes.rst]

As well as the `#`, another way of creating a comment is to surround the comment with triple quotes (`""" """` or `''' '''`). A triple quotes comment can be spread over multiple lines.

121.22.1 comments_multiline.py

Write and test a triple quote program, `comments_multiline.py`.

```
""" Triple quotes
 for multi-line comments """
```

121.22.2 comments.py

Predict then test the output of this code.

```
""" One, two,
 three, """
print ("four, five", "seven") # eight
# nine
print("10", """ eleven
12""")
```

121.23 Classes - methods with parameters [class-03-params.rst]

In a similar manner to functions, methods can take arguments.

121.23.1 method_parameter.py

```
class ClassMethodWithParameter:
    def times(self, val_1, val_2):
        print(val_1 * val_2)

    def add(self, val_1, val_2):
        print(val_1 + val_2)

instance = ClassMethodWithParameter()
instance.times(3, "hi ")
instance.add(1.2, 2)
```

121.24 NumPy [projects-16-numpy.rst]

NumPy is a package for scientific computing with Python. NumPy usually operates on n-dimensional arrays with elementwise operations. Thus when you multiply a Numpy array using the * operator it is **not** mathematical matrix multiplication.

Experiment with the following in interactive mode.

121.24.1 Import Numpy

```
>>> import numpy as np
```

121.24.2 Adding two arrays

```
>>> a = np.array([2,3,4])
>>> b = np.array([1,2,3])
>>> a + b
```

121.24.3 Raise to the power of 3

```
>>> a**3
```

121.24.4 Create an array and reshape it

```
>>> a = np.arange(10).reshape(2, 5)
>>> a
```

121.24.5 Check its shape

```
>>> a.shape
```

121.24.6 How many dimensions

```
>>> a.ndim
```

121.24.7 What is the type of the array elements

```
>>> a.dtype.name
```

121.24.8 Number of elements of the array

```
>>> a.size
```

121.24.9 What is the type of the array

```
>>> type(a)
```

121.25 NumPy and matrices [numpy_matrix_multiplication.rst]

There are several ways to do matrix multiplication in NumPy. One is to use the `@` operator. Try the following.

121.25.1 matrix_mul.py

```
import numpy as np

a = np.array([
    [1, 2],
    [3, 4]])

b = a @ a

print(b)
```

121.25.2 matrix_mul_trans.py

Comment this to explain what is going on.

```
import numpy as np

a = np.array([
    [1, 2],
    [3, 4]])

b = a.transpose()

c = a @ b

print(c)
```

121.25.3 matrix_det.py

From your maths classes, you might remember **matrix determinants**. Determinants are a way of characterising square matricies. You can use a Numpy (specifically the linear algebra package) to calculate determinants, however, the method below may not be suitable for large matricies (see the Numpy docs for more information).

```
import numpy as np

a = np.array(
    [[5, 2],
     [3, 4]])

print(np.linalg.det(a))
```

121.26 Project - plotting example [projects-13.rst]

121.26.1 plot_heat.py

Write and test this code.

```
import numpy as np
import matplotlib.pyplot as plt

# Generate some test data
x = np.random.randn(8873)
y = np.random.randn(8873)

heatmap, xedges, yedges = np.histogram2d(x, y, bins=50)
extent = [xedges[0], xedges[-1], yedges[0], yedges[-1]]

plt.clf()
plt.imshow(heatmap, extent=extent)
plt.show()
```

Original code

http://stackoverflow.com/questions/2369492/generate-a-heatmap-in-matplotlib-using-a-scatter-data-set

121.26.2 plot_heat_cmts.py

With the help of the investigate, instigate and comment your previous code and save it as `plot_heat_cmts.py`.

121.27 Waypoint [preamble-10-waypoint.rst]

Well done at getting this far through the work. Already, you know enough Python to put it to some practical use. In the following sections you will reinforce what you have learned, extend your knowledge and learn some new concepts. There will be repetition, however, do not skip any sections and go at a steady pace. Even if revision, it will still be excellent practice.

CHAPTER

TWO

EXERCISES 3

122.1 Strings [string-03.rst]

122.1.1 doing.py

Write and test the following program.

```
string_variable = "What am I doing?\n"
print(5 * string_variable)
```

122.1.2 doing_comments.py

Copy doing.py, thoroughly comment and save as doing_comments.py.

122.1.3 doing_badly.py

Copy doing.py, saving as doing_badly.py remove the \n, run and explain in comments what is going on.

122.1.4 Functions - parameters, functions_and_strings.py

Write and test the following program, functions_and_strings.py.

```
def function_two(argument1, argument2):
    print("argument 1: %s, argument 2: %s" % (argument1, argument2))

def function_one(argument):
    print("argument: %s" % argument)

def function_nothing():
    print("Zero.")

function_two("Two","One")
function_one("One")
function_nothing()
```

122.2 Numbers [var-04.rst]

122.2.1 pounds_dirham_convert.py

Write program of only a few lines to convert British Pounds into Moroccan Dirham and vice versa.

122.2.2 convert1234.py

Write a short program, of only a few lines, that converts the floating point number **123.4** into an integer.

122.2.3 Operators - numerical average_of_four.py

- Write a very short program, only a few lines, to calculate the average of 5, 102, -12.2 and 5, `average_of_four.py`.

122.3 if revision [if-06-revision.rst]

122.3.1 zombie_escape.py

Project - More *if* practice - zombie escape game. Write, test and comment the following program, save as **zombie_escape.py**.

```
print("Terrible moaning at your front door.")
print("There are two windows.")
print("Do you get out through window #1")
print("or window #2 to the back?")
window = input("> ")

if window == "1":
    print("There's a large Zombie here.")
    print("What do you do?")
    print("1. Try to cure the ailment.")
    print("2. Shout at the Zombie.")
    large = input("> ")

    if large == "1":
        print("The Zombie eats your face off.")
        print("Just brilliant!")
    elif large == "2":
        print("The Zombie eats your legs off.")
        print("Just brilliant!")
    else:
        print("Well, doing %s is probably better." % large)
        print("Zombie stumbles away.")
elif window == "2":
    print("Here's a Zombie horde.")
    print("1. Spin.")
    print("2. Jump.")
    print("3. Hope.")
```

(continues on next page)

(continued from previous page)

```
    horde = input("> ")

    if horde== "1" or horde == "2":
        print("You survive by...")
        print("covering yourself in chocolate.")
        print("Just brilliant!")
    else:
        print("You become a Zombie and rot.")
        print("Just brilliant!")
else:
    print("You freak and die of heart failure.")
    print("Just brilliant!")
```

122.4 Looping revision [loop-06.rst]

Try to figure out the answer to each question before you run the code.

122.4.1 list_method_join.py

What it the output of this piece of code?

```
# list_method_join.py

five_list = [1,2,3,4,5]
six_nine_list = [6,7,8,9]
joined_list = five_list + six_nine_list
print("joined_list = ", joined_list)
```

122.4.2 while_not_q.py

Write a program that uses a **while** loop to continually take command line input from a user. The program will exit when a **q** is input or print a suitable message.

122.5 Numbers [func-06-param-ctd.rst]

122.5.1 sum_3.py

Write and call a function **sum** that that takes 3 arguments and returns the result of adding them together.

122.5.2 bigger.py

Create a program which has a function `test_bigger(a, b)` which returns `True` if `a` is greater than `b` otherwise return `False`. This program should also have test code.

122.6 Modules - if __name__ == "__main__": [module-04.rst]

Often the start point of a program is contained in a `if __name__ == "__main__":` block at the bottom of the a Python source file. This block is only ever used in the top level source file. In all the other files below this level anything in this block is ignored. Thus this is a good place to put code that can be used for testing or code that you do not want to be imported if a top level source file were used as a sub-module.

122.6.1 two_functions_1.py

Write and run this program, `two_functions_1.py`.

```
# two_functions_1.py

def fun1():
    return 1

def fun2():
    return 2

a = fun1()
b = fun2()
c = fun1()
print(a + b + c)
```

122.6.2 two_functions_2.py

Write run this program, `two_functions_2.py`.

```
# two_functions_2.py

def fun1():
    return 1

def fun2():
    return 2

if __name__ == "__main__":
    a = fun1()
    b = fun2()
    c = fun1()
    print(a + b + c)
```

122.6.3 Importing

Explain in comments what is going with these two programs.

122.6.4 main_1.py

Write and test this program, `main_1.py`.

```
# main_1.py

from two_functions_1 import *

print(fun1())
```

122.6.5 main_2.py

Write and test this program, `main_2.py`.

```
# main_2.py

from two_functions_2 import *

print(fun1())
```

122.7 Errors - assert [except-04-assert.rst]

assert conditionally raises and exception - often used in debugging and development.

```
>>> assert True, "Assert error"


>>> assert False, "Assert error"
```

122.8 Classes __init__ operator overloading [class-04-overload.rst]

Particular combinations of underscores at the beginning and/or end of names in your program can have special meanings. The double underscore at the beginning and end of a class method name (**operator overload methods**) are typically reserved for built-in methods or variables. `__init__` is used in a class to initiate an object.

122.8.1 initialisation_of_class.py

Write and test the following code, initialisation_of_class.py.

```
class InitClass:
    def __init__(self):
        print ("Instance")
    def method(self):
        print("Like a function.")

instance_1 = InitClass()
instance_1.method()
instance_1.method()

instance_2 = InitClass()
instance_2.method()
instance_2.method()
```

122.8.2 initialisation_of_class_ott_comments.py

Make a copy of your previous code and save it as initialisation_of_class_ott_comments.py and add a comment to each line explaining what it does.

122.8.3 double_class.py

Classes - updating internal attribute

Write and test this program which uses a method to update an attribute.

```
class DoubleClass:
    def __init__(self, val_in):
        self.value = val_in

    def double_value(self):
        self.value = 2 * self.value


instance_of_double_class = DoubleClass(1)
print(instance_of_double_class.value)

instance_of_double_class.double_value()
print(instance_of_double_class.value)

instance_of_double_class.double_value()
print(instance_of_double_class.value)

instance_of_double_class = DoubleClass("Hello ")
instance_of_double_class.double_value()
print(instance_of_double_class.value)
```

122.8.4 double_class_ott_comments.py

To demonstrate your understanding, fully comment the code then save as double_class_ott_comments.py.

122.9 NumPy and matrices [numpy_linear_equations.rst]

Use NumPy to write programs to solve the system of equations:

```
x  + 2y = 5
3x + 4y = 6
```

122.9.1 numpy_solve_inv.py

This uses the inv(erse) and dot methods. This is only reliable if the system is **well** behaved.

```
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.array([5, 6])

x = np.linalg.inv(a).dot(b)

print(x)
print(a @ x)
```

122.9.2 numpy_solve.py

This uses the solve function and will be more reliable that the previous program.

```
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.array([5, 6])

x = np.linalg.solve(a, b)

print(x)
print(a @ x)
```

122.10 Waypoint [preamble-10-waypoint.rst]

Well done at getting this far through the work. Already, you know enough Python to put it to some practical use. In the following sections you will reinforce what you have learned, extend your knowledge and learn some new concepts. There will be repetition, however, do not skip any sections and go at a steady pace. Even if revision, it will still be excellent practice.

122.11 Help 3 [help-05-pydoc.rst]

122.11.1 pydoc

While the internet and search engines are the first place you will look for Python help and information, pydoc, a module for generating documentation, is another option. In fact, the interactive help you used earlier, uses pydoc behind the scenes.

You can run ‘pydoc’ from your computer to view the help from outside Python’s interactive mode. From your computer’s command line, try this.

```
> pydoc sys
```

122.12 Strings - r, another method, avoiding \, to escape quotes [string-06-raw_strings.rst]

Using **raw strings**.

- String literals may be prefixed by r or R.
- The string is then a **raw string**.
- **Raw strings** have different rules for escape characters.

122.12.1 string_escape_r.py

Write and test this program.

```
print("C:\python\number")
print(r"C:\python\number")
```

122.12.2 string_escape_r_comment.py

Make a copy of the previous program and write code comments explaining what is going on.

122.13 Functions - wrapping a while loop into a function [loop-07.rst]

122.13.1 simple_while_loop.py

If you have not written this code already, write and test:-

```
# simple_while_loop.py

x = 0
while x < 10:
    x = x + 1
    print(x)
```

122.13.2 while_loop_function.py

Copy and convert simple_while_loop.py, into a function that you can call with a variable that replaces the **10**. Save your program as while_loop_function.py. Call your new function from within your program with different numbers.

122.13.3 step_size.py

By adding another variable to the function *parameter*, that lets you change the step size (currently + 1), change your function so it increments by different amounts. Save this as step_size.py and test this function to see what effect your changes have made.

122.14 Functions - optional/default parameters [func-07-param-default.rst]

Functions parameters can have default values which are used if no argument is provided by the call. For this exercise, use your numbers directory.

122.14.1 geometric_score.py

Write, fix and test the following code, **geometric_score.py**.

```
# geometric_score.py

def increase(previous_score, ratio = 0.1):
    difference = 1.0 - previous_score
    increase_amount = difference * ratio
    score = previous_score + increase_amount
    return score

def decrease(previous_score, ratio = 0.1):
    decrease_amount = previous_score * ratio
    score = previous_score - decrease_amount
    return score

score = 0.5
while(score < 0.9):
    score = increase(score)
    print(score)
```

122.14.2 geometric_score_comments.py

By expanding the calling code, for an initial score between 0.0 and 1.0 explain how the functions work. Save this code as **geometric_score_comments.py**.

122.15 Errors 4 - try, except, else and finally [except-05-else_finally.rst]

There are four statements which are concerned with exceptions and exception handling:

```
try:
    # primary action
    pass
except:
    # if any exception raised
    pass
else:
    # if no exception raised
    pass
finally:
    # Always perform these clean up actions
    # whether an exception occurs or not.
    pass
```

122.15.1 finally_1.py

What is the result of this simple error handling?

```
# finally_1.py

try:
    1/0
except:
    msg = 'This message.'
finally:
    msg = 'That message.'

print("What message: ", msg)
```

122.16 Classes - operator overloading __add__ and __sub__ [class-05-overload-add_sub.rst]

If appropriate, you can make the + and - operators work with your objects with the methods

> __add__ makes the + work.
>
> __sub__ makes the - work.

122.16.1 celsius.py

Write and test this program `celsius.py`.

```
# `celsius.py`

class Celsius:
    def __init__(self, temp_in):
        self.temp = temp_in
    def add(self, rhs):
        new_temp_value = self.temp + rhs.temp
        return Celsius(new_temp_value)

temp_1 = Celsius(32)
temp_2 = Celsius(100)
temp_3 = temp_1.add(temp_2)
temp_3.temp
print(temp_1.temp, temp_2.temp, temp_3.temp)
```

122.16.2 celsius_overload_01.py

Write and test this program `celsius_overload_01.py`. This replaces the add method of `celsius.py` with `__add__`.

```
# celsius_overload_01.py

class Celsius:
    def __init__(self, temp_in):
        self.temp = temp_in
    def __add__(self, rhs):
        new_temp_value = self.temp + rhs.temp
        return Celsius(new_temp_value)
    def subtract(self, rhs):
        new_temp_value = self.temp - rhs.temp
        return Celsius(new_temp_value)

temp_1 = Celsius(32)
temp_2 = Celsius(100)
temp_3 = temp_1 + temp_2
print(temp_1.temp, temp_2.temp, temp_3.temp)
```

122.16.3 celsius_overload_02.py

Change your `celsius_overload_01.py` program so that the following `celsius_overload_02.py` will work.

```
# celsius_overload_02.py
temp_1 = Celsius(32)
temp_2 = Celsius(100)
temp_3 = temp_1 + temp_2 - temp_2 - temp_2
print(temp_1.temp, temp_2.temp, temp_3.temp)
```

122.17 Functions 3.2, Scope 2 and Variables - `global` [func-09-scope-global.rst]

Things that you create inside a function are only accessible inside a function. The opposite is not true, you can, if the conditions are right, access things sitting outside of a function from within a function without them being passed in as an argument. However, despite being able to access, there can be complications with changing an external object. If you do want to change something outside the function you will have to use **global**. Try to avoid overusing **global**.

Global variables should be used sparingly.

122.17.1 scope_03.py

```
# scope_03.py

x = 1 # Global (file) scope.

def scope_function():
    global x
    x = 2     # Global (file) scope.
    print(x)
print(x)
scope_function()
print(x)
```

122.17.2 scope_04.py

Before running, predict what the output of this program will be. You must understand this before moving on.

```
# scope_04.py

x = 1
print(x)
def fun1():
    print('fun1')
    print(x)

def fun2():
    print('fun2')
    x = 100
    print(x)

print(x)
fun1()
fun2()
x += 1
print(x)
fun1()
fun2()
```

122.17.3 scope_05.py

Write this new version of the previous code. After running, fully explain, in comments, what is going on.

```
# scope_05.py

x = 1
print(x)
def fun1():
    print('fun1')
    print(x)

def fun2():
    print('fun2')
    global x
    x = 100
    print(x)

print(x)
fun1()
fun2()
x += 1
print(x)
fun1()
fun2()
```

122.17.4 scope_06.py

What is the output of this code?

```
# scope_06.py

number = 0

def func():
    global number
    number += 1

func()
func()
func()
print(number)
```

122.18 Modules - os module [module-05.rst]

The os module provides functions for interacting with your operating system.

122.18.1 simple_system.py

Use help and dir to investigate os interactively then try this short example and fully comment.

```
#simple_system.py

from os import getcwd, chdir

cwd = getcwd() # Return the current working directory
print(cwd)

chdir('..') # Change directory
cwd = getcwd()
print(cwd)
```

122.19 SymPy[sympy.rst]

Symbolic computation concerns calculations using mathematical symbolically rather than numbers. The results are exact and remain in symbolic form until they are forced into a numerical representation.

122.19.1 math module

Try the following in interactive mode.

```
import math
```

```
math.sqrt(2)
```

```
2 * math.sqrt(2)
```

```
math.sqrt(8)
```

122.19.2 sympy module

Try the following in interactive mode.

```
import sympy
```

```
sympy.sqrt(2)
```

```
2 * sympy.sqrt(2)
```

```
sympy.sqrt(8)
```

122.19.3 sympy display

Again, in interactive mode, type the following.

```
sympy.init_printing(use_unicode=True)
```

```
sympy.sqrt(2)
```

```
2 * sympy.sqrt(2)
```

```
sympy.sqrt(8)
```