2_Coding

September 25, 2016

```
In [1]: from notebook.services.config import ConfigManager
        from IPython.paths import locate_profile
        cm = ConfigManager(profile_dir=locate_profile(get_ipython().profile))
        cm.update('livereveal', {
                      'theme': 'solarized',
                      'transition': 'zoom',
                      'start_slideshow_at': 'selected',
        })

Out[1]: {'start_slideshow_at': 'selected', 'theme': 'solarized', 'transition': 'zoc
```

1 Coding in Python

1.1 Dr. Chris Gwilliams

1.1.1 gwilliamsc@cardiff.ac.uk

2 Writing in Python: PEP

2.1 Python Enhancement Proposals

Unsure how your code should be written? PEP is a style guide for Python and provides details on what is expected.

- Use 4 spaces instead of tabs
- Lines should be 79 characters long
- Variables should follow `snake_case`
  - All lower case words, separated by underscores (_)
- Classes should be Capitalised Words (`MyClassExample`)

PEP

3 Comments

Sometimes, you need to describe your code and the logic may be a bit complicated, or it took you a while to figure it out and you want to make a note.

You can't just write some text in the file or you will get errors, this is where comments come in! Comments are descriptions that the Python interpreter ignores.

Just type a # amd what ever you want to write and voíla!

It is ALWAYS a good idea to comment your code!

```
In [2]: # Does this make sense without comments?
        with open('myfile.csv', 'rb') as opened_csv:
            spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
            for row in spamreader:
                print (', '.join(row))


        ---------------------------------------------------------------------------

        FileNotFoundError                         Traceback (most recent call last)

        <ipython-input-2-04109d93ef87> in <module>()
          1 # Does this make sense without comments?
    ----> 2 with open('myfile.csv', 'rb') as opened_csv:
          3     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
          4     for row in spamreader:
          5         print (', '.join(row))


        FileNotFoundError: [Errno 2] No such file or directory: 'myfile.csv'


In [2]: # How about this?

        #open csv file in readable format
        with open('myfile.csv', 'rbU') as opened_csv:
            # read opened csv file with spaces as delimiters
            spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
            # loop through and print each line
            for row in spamreader:
                print (', '.join(row))

/usr/local/lib/python3.4/site-packages/ipykernel/__main__.py:4: DeprecationWarning:



        ---------------------------------------------------------------------------

        FileNotFoundError                         Traceback (most recent call last)

        <ipython-input-2-9cc1b544c150> in <module>()
          2 
          3 #open csv file in readable format
    ----> 4 with open('myfile.csv', 'rbU') as opened_csv:
```

```
    # read opened csv file with spaces as delimiters
    spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
```

```
FileNotFoundError: [Errno 2] No such file or directory: 'myfile.csv'
```

4 Types

Python has a type system (variables have types), even if you do not specify it when you declare them.

- String
- Float
- Boolean
- Integer
- None

4.0.1 Exercise

Give me an example of each of these types.
What is the None type?
Use the Internet to find me examples of 1 other type in Python
We will come back to type as the course progresses.

5 Literals

Literally a value.
All of the examples you just gave are literals.

```
In [2]: "Gavin" #String Literal
        4 #Integer Literal
        3.14 #??? Literal
```

```
Out[2]: 3.14
```

3.14 is a `float` literal

6 Variables

Literals are all well and good for printing but what about when we need to change and store these literals?
Variables are ways of giving literal values a name in order to refer to them later.
Below, we are **declaring** and **instantiating** a variable

```
In [ ]: # declared and instantiated
        name = "Gavin"

        # declared, but not instantiated
        new_name = None
```

7 Variables in Python

Technically, Python does not have `empty` variables that are not instantiated. A variable exists as soon as it is assigned a value. Like this:

```
In [1]: x # does not exist so cannot print it


        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-1-401b30e3b8b5> in <module>()
    ----> 1 x


        NameError: name 'x' is not defined
```

```
In [ ]: x = 1
        print(x)
```

8 Variables in Python

In many languages, when a variable is instantiated, it is reserved into a block of memory and the variable points to that memory address, often unique for that variable.

In Python, it is more like that memory address is tagged with the variable name. So, if we create three variables that have the same literal value, then they all point to the same memory address. Like so:

```
In [6]: a = 1
        b = 1
        c = 1
        print(id(a))
        print(id(b))

4406331952
4406331952
```

9 id and Multi Variable Assignment

The id function is built into Python and returns the memory address of variables provided to it.
This is easier to see when we use multi-variable assignment in Python:

```
In [1]: a,b,c = 1,1,1
        a,b,c = "Name",12,"6ft"
        print(a,b,c) #NOTE: always balance the left and the right. 5 variables must
```

```
('Name', 12, '6ft')
```

10 Don't Call It That:

These are keywords reserved in Python, so do **not** name any of your variables after these! You will learn about what many of these do throughout this course.

False	class	finally	is	return
continue	for	lambda	try	True
def	from	nonlocal	while	and
del	global	not	with	as
elif	if	or	yield	assert
else	import	pass	break	except
in	raise	None		

11 Useful links on Python variables

http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html#other-languages-have-variables
http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/variables.html
http://foobarnbaz.com/2012/07/08/understanding-python-variables/
http://www.diveintopython.net/native_data_types/declaring_variables.html

12 Types II

Python is *Strongly Typed* - The Python interpreter keeps track of all the variables and their associated types.
AND
Python is *Dynamically Typed* - Variables can be reassigned from their types. A variable is simply a value bound to a name, the variable does not hold a type, only the value does.

```
In [6]: print("Hello " + "World") #ok
        print("hello" + 5) #strongly typed means this cannot happen!
        name = "Chris"
        name = "Pi"
        "pi" + 6 #Strongly typed means no adding different types together!
        name = 3.14 #dynamically typed means yes to changing the type of a variable
```

```
Hello World



        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-6-16cc78780c92> in <module>()
          1 print("Hello " + "World") #ok
    ----> 2 print("hello" + 5) #strongly typed means this cannot happen!
          3 name = "Chris"
          4 name = "Pi"
          5 name + 6 #Strongly typed means no adding different types together!


        TypeError: Can't convert 'int' object to str implicitly
```

13 Finding Types

Got some code and don't know what the types are?
Python has some functions to help with this.

```
In [ ]: type('am I your type?')
```

13.0.1 Exercise

Try this with different types, how many can you find?
What happens when you do `type([])`?
`type` is an example of a built-in function, we will come back to these in a few sessions.

14 Strings

Strings in Python are unlike many languages you have seen before.
They can be:

```
In [ ]: 'single_quotes'
```

What do single quotes usually mean in most languages?
Strings wrapped in 'single quotes' are typically `chars` (single characters). Python does not have this type.

```
char yes = 'Y' //char (Python does not have this)
string no = "no" //string
```

What are chars used for? What does Python have instead?

Chars are typically used as single character flags, like a 'Y' or an 'N' as an answer to a question, or to hold an initial.

Anything text based can be stored in a string but flags can be represented as a 0 or 1 or even using a Boolean value,, which is easiest to check against.

What happens if you use a single quote for strings and you write the word `don't` in the string? Try it out now!

How do we get around that?

15 Escaping Strings

When you want to include special characters (like `'`) then it is always good to `escape` them!

Ever seen a new line written as `\n`? That is an example of escaping.

15.1 Escape Character

```
An escape character is a character which invokes an alternative interpretation on s
```

This is pretty much always \

```
In [ ]: 'isn't # not gonna work
```

```
In [ ]: 'isn\'t' # works a charm!
```

16 Strings II

If you do not want to escape every special character, maybe there is a better way?

```
"I am a string and I don't care what is written inside me"
"""I am a string with triple double quotes and I can
run across multiple lines"""
```

Double quotes is generally better as you do not have to escape these special characters.

16.0.1 Exercise

Declare a `variable` to store a `boolean literal`

Declare and instantiate a new `variable` that stores your age

The first one was a trick! Declaring a new variable means giving it no value!

The second one would be: `age = 20`

17 Reassigning Variables

It would not be fun to have to create a new variable for every thing you want to store, right?

As well as being a huge inconvenience, it is actually really inefficient.

```
age = 40
# 1 year passes
age = 41
```

Easy, right? By using the same variable name, it is now associated with your new value and the old value will be cleared up.

18 Printing

We have seen this `print` keyword thrown around alot, right? This is the best way to show some information. Especially useful if your script takes a long time to run!

E.g. `print("stuff")`

```
In [5]: print("Got something to say?")
        print("Use the print statement")
        print("to print a string literal")
        print("float literal")
        print(3.14)
        more_string = "or variable"
        print(more_string)
```

```
Got something to say?
Use the print statement
to print a string literal
float literal
3.14
or variable
```

18.0.1 Exercise

Create a variable of type `string` and then `reassign` it to a float literal.

Now try adding a `float` literal to your variable

19 Operators

What is an operator? What do you remember from maths?

- + (add)
- − (subtract)
- / (divide)
- ⋆ (multiply)

There are more, but we will get to them!

19.0.1 Exercise

- Add 3 and 4
- Add 3.14 + 764 (what is the difference to the above answer?)
- Subtract 100 from 10
- Multiply 10 by 10
- Add 13 to 'adios'
- Multiply 'hello' by 5
- Divide 10 by 3
- Divide 10 by 3 but use 2 / (what happens?)

In Python 3, // is floor division and / is floating point division

20 Adding/Dividing/Subtracting Across Types

```
In [ ]: float_type = 3.0
        int_type = 5
        print(int_type + float_type)

        string_type = "hello"
        print(string_type + float_type) # what is the error?

        bool_type = True
        print(string_type + bool_type)

        print(int_type + bool_type) # does this work? Why?
```

21 Multiplying Across Types

While Python is not happy to add/divide/subtract numbers from strings, it is more than happy to multiply

```
In [ ]: float_type = 3.0
        int_type = 5
        print(int_type * float_type)

        string_type = "hello"
        print(string_type * int_type)

        bool_type = True
        print(string_type * bool_type) #why does this work?

        print(int_type * bool_type)
```

22 Operating and Assigning

You may have a variable and want to change the value, this is reassigning, right?

```
year = 1998
year = 1999
```

There has to be an easier way. THERE IS. What is it?

```
In [ ]: year = year + 1
        year += 1
```

22.0.1 Exercise

Try this with all the operators you know.

23 Built in Functions

A function is a block of code that: - receives an input(s) (also known as arguments) - performs an operation (or operations) - Optionally, returns an output
Python has some built-in functions and we have used one already. What was it?

24 Functions - Print

```
print("Hello")
```

Format: - function_name - Open brackets - inputs (separated, by, commas) - Close brackets
Sometimes, inputs are optional. Not always. We will get to this.

25 Other Built in Functions

- `str()`
- `len()`
- `type()`
- `int()`

25.0.1 Exercise

Find out what the above functions do and use them in a script
What happens when you don't give each an argument?
Look up and write up definitions for the `id` and `isinstance` functions
str - converts an object to a string type len - prints the length of an object type - tells you the type of a literal or a variable int - converts a type to an integer
id - a unique id that relates to where the item is stored in memory isinstance - checks if an object if of the supplied type

26 Functions to Help You

26.1 dir

```
In [ ]: dir("")
```

26.2 help

```
In [1]: help(int)
        help("")
        help(1)
        name = "terry"
        help(name)
```

```
Help on class int in module builtins:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __ceil__(...)
 |      Ceiling of an Integral returns itself.
 |
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __float__(self, /)
 |      float(self)
 |
 |  __floor__(...)
 |      Flooring an Integral returns itself.
 |
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |
 |  __format__(...)
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getnewargs__(...)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __index__(self, /)
 |      Return self converted to an integer, if self is suitable for use as an inde
 |
 |  __int__(self, /)
 |      int(self)
 |
 |  __invert__(self, /)
 |      ~self
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __lshift__(self, value, /)
 |      Return self<<value.
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __neg__(self, /)
 |      -self
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __pos__(self, /)
 |      +self
 |
 |  __pow__(self, value, mod=None, /)
 |      Return pow(self, value, mod).
 |
 |  __radd__(self, value, /)
 |      Return value+self.
 |
 |  __rand__(self, value, /)
 |      Return value&self.
 |
 |  __rdivmod__(self, value, /)
 |      Return divmod(value, self).
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __rfloordiv__(self, value, /)
 |      Return value//self.
 |
 |  __rlshift__(self, value, /)
 |      Return value<<self.
 |
 |  __rmod__(self, value, /)
 |      Return value%self.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __ror__(self, value, /)
 |      Return value|self.
 |
 |  __round__(...)
 |      Rounding an Integral returns itself.
```

```
|      Rounding with an ndigits argument also returns an integer.
|
|  __rpow__(self, value, mod=None, /)
|      Return pow(value, self, mod).
|
|  __rrshift__(self, value, /)
|      Return value>>self.
|
|  __rshift__(self, value, /)
|      Return self>>value.
|
|  __rsub__(self, value, /)
|      Return value-self.
|
|  __rtruediv__(self, value, /)
|      Return value/self.
|
|  __rxor__(self, value, /)
|      Return value^self.
|
|  __sizeof__(...)
|      Returns size in memory, in bytes
|
|  __str__(self, /)
|      Return str(self).
|
|  __sub__(self, value, /)
|      Return self-value.
|
|  __truediv__(self, value, /)
|      Return self/value.
|
|  __trunc__(...)
|      Truncating an Integral returns itself.
|
|  __xor__(self, value, /)
|      Return self^value.
|
|  bit_length(...)
|      int.bit_length() -> int
|
|      Number of bits necessary to represent self in binary.
|      >>> bin(37)
|      '0b100101'
|      >>> (37).bit_length()
|      6
|
|  conjugate(...)
```

```
 |      Returns self, the complex conjugate of any int.
 |  
 |  from_bytes(...) from builtins.type
 |      int.from_bytes(bytes, byteorder, *, signed=False) -> int
 |      
 |      Return the integer represented by the given array of bytes.
 |      
 |      The bytes argument must either support the buffer protocol or be an
 |      iterable object producing bytes.  Bytes and bytearray are examples of
 |      built-in objects that support the buffer protocol.
 |      
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order value.
 |      
 |      The signed keyword-only argument indicates whether two's complement is
 |      used to represent the integer.
 |  
 |  to_bytes(...)
 |      int.to_bytes(length, byteorder, *, signed=False) -> bytes
 |      
 |      Return an array of bytes representing an integer.
 |      
 |      The integer is represented using length bytes.  An OverflowError is
 |      raised if the integer is not representable with the given number of
 |      bytes.
 |      
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order value.
 |      
 |      The signed keyword-only argument determines whether two's complement is
 |      used to represent the integer.  If signed is False and a negative integer
 |      is given, an OverflowError is raised.
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |  
 |  denominator
 |      the denominator of a rational number in lowest terms
 |  
 |  imag
 |      the imaginary part of a complex number
 |  
 |  numerator
 |      the numerator of a rational number in lowest terms
 |
 |  real
 |      the real part of a complex number


Help on int object:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __ceil__(...)
 |      Ceiling of an Integral returns itself.
 |
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __float__(self, /)
 |      float(self)
 |
 |  __floor__(...)
 |      Flooring an Integral returns itself.
 |
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |
 |  __format__(...)
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getnewargs__(...)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __index__(self, /)
 |      Return self converted to an integer, if self is suitable for use as an inde
 |
 |  __int__(self, /)
 |      int(self)
 |
 |  __invert__(self, /)
 |      ~self
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __lshift__(self, value, /)
 |      Return self<<value.
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
```

```
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __neg__(self, /)
 |      -self
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __pos__(self, /)
 |      +self
 |
 |  __pow__(self, value, mod=None, /)
 |      Return pow(self, value, mod).
 |
 |  __radd__(self, value, /)
 |      Return value+self.
 |
 |  __rand__(self, value, /)
 |      Return value&self.
 |
 |  __rdivmod__(self, value, /)
 |      Return divmod(value, self).
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __rfloordiv__(self, value, /)
 |      Return value//self.
 |
 |  __rlshift__(self, value, /)
 |      Return value<<self.
 |
 |  __rmod__(self, value, /)
 |      Return value%self.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __ror__(self, value, /)
 |      Return value|self.
 |
 |  __round__(...)
 |      Rounding an Integral returns itself.
```

```
|      Rounding with an ndigits argument also returns an integer.
|
|  __rpow__(self, value, mod=None, /)
|      Return pow(value, self, mod).
|
|  __rrshift__(self, value, /)
|      Return value>>self.
|
|  __rshift__(self, value, /)
|      Return self>>value.
|
|  __rsub__(self, value, /)
|      Return value-self.
|
|  __rtruediv__(self, value, /)
|      Return value/self.
|
|  __rxor__(self, value, /)
|      Return value^self.
|
|  __sizeof__(...)
|      Returns size in memory, in bytes
|
|  __str__(self, /)
|      Return str(self).
|
|  __sub__(self, value, /)
|      Return self-value.
|
|  __truediv__(self, value, /)
|      Return self/value.
|
|  __trunc__(...)
|      Truncating an Integral returns itself.
|
|  __xor__(self, value, /)
|      Return self^value.
|
|  bit_length(...)
|      int.bit_length() -> int
|
|      Number of bits necessary to represent self in binary.
|      >>> bin(37)
|      '0b100101'
|      >>> (37).bit_length()
|      6
|
|  conjugate(...)
```

```
 |      Returns self, the complex conjugate of any int.
 |
 |  from_bytes(...) from builtins.type
 |      int.from_bytes(bytes, byteorder, *, signed=False) -> int
 |
 |      Return the integer represented by the given array of bytes.
 |
 |      The bytes argument must either support the buffer protocol or be an
 |      iterable object producing bytes.  Bytes and bytearray are examples of
 |      built-in objects that support the buffer protocol.
 |
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order value.
 |
 |      The signed keyword-only argument indicates whether two's complement is
 |      used to represent the integer.
 |
 |  to_bytes(...)
 |      int.to_bytes(length, byteorder, *, signed=False) -> bytes
 |
 |      Return an array of bytes representing an integer.
 |
 |      The integer is represented using length bytes.  An OverflowError is
 |      raised if the integer is not representable with the given number of
 |      bytes.
 |
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order value.
 |
 |      The signed keyword-only argument determines whether two's complement is
 |      used to represent the integer.  If signed is False and a negative integer
 |      is given, an OverflowError is raised.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  numerator
 |      the numerator of a rational number in lowest terms
 |
 |  real
 |      the real part of a complex number

no Python documentation found for 'terry'
```

coding

26.2.1 Homework

Complete sheet `2_Coding`