Contents

Python object and data structures Basics

Python Comparison Operators

Python Statements

Methods and Functions

Modules and Packages

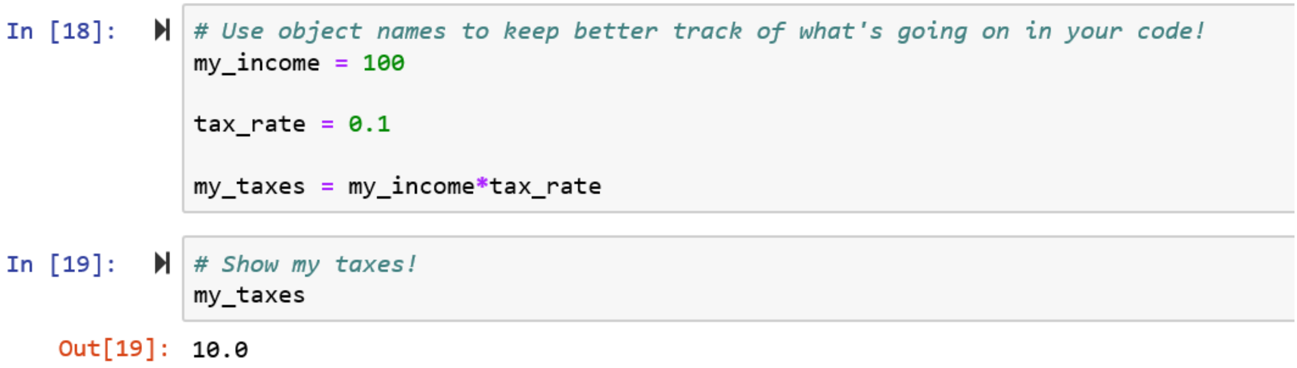
Errors and Exception Handling

Built-in Functions

Python Object and Data Structure Basics

Variable Assignment

- Names can not start with a number
- Names can not contain spaces, use underscore (_) instead
- Names can not contain any of these symbols: **:'",<>/?|\!@#%^&*~-+**
- It's considered best practice (PEP8) that names are lowercase with underscores
- Avoid using Python built-in keywords like list and str
- Avoid using the single characters l (lowercase letter el), O (uppercase letter oh) and I (uppercase letter eye) as they can be confused with 1 and 0.
- Example:

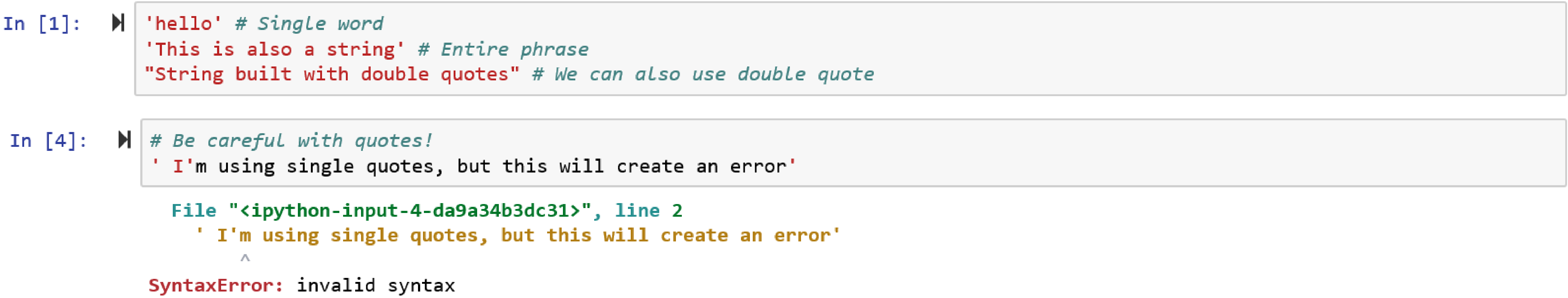


```
In [18]:  # Use object names to keep better track of what's going on in your code!
          my_income = 100

          tax_rate = 0.1

          my_taxes = my_income*tax_rate

In [19]:  # Show my taxes!
          my_taxes

Out[19]:  10.0
```

Python Object and Data Structure Basics

Strings

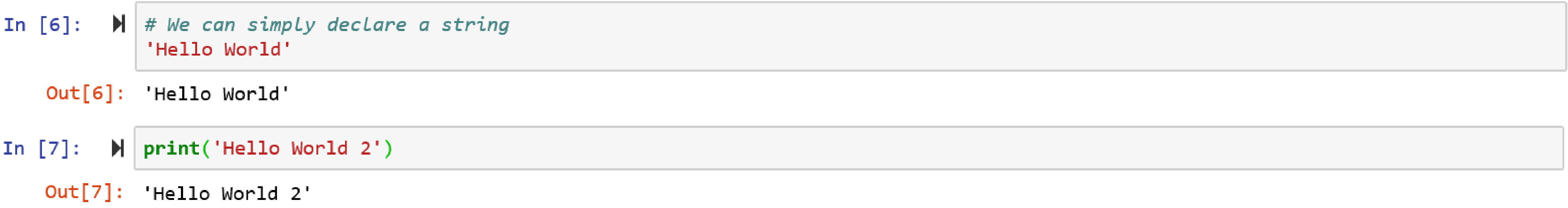
- **Creating Strings:**



```
In [1]:  'hello' # Single word
         'This is also a string' # Entire phrase
         "String built with double quotes" # We can also use double quote
```

```
In [4]:  # Be careful with quotes!
         ' I'm using single quotes, but this will create an error'

           File "<ipython-input-4-da9a34b3dc31>", line 2
             ' I'm using single quotes, but this will create an error'
                ^
         SyntaxError: invalid syntax
```

- **Printing Strings**



```
In [6]:  # We can simply declare a string
         'Hello World'

Out[6]:  'Hello World'
```

```
In [7]:  print('Hello World 2')

Out[7]:  'Hello World 2'
```

Python Object and Data Structure Basics

Strings

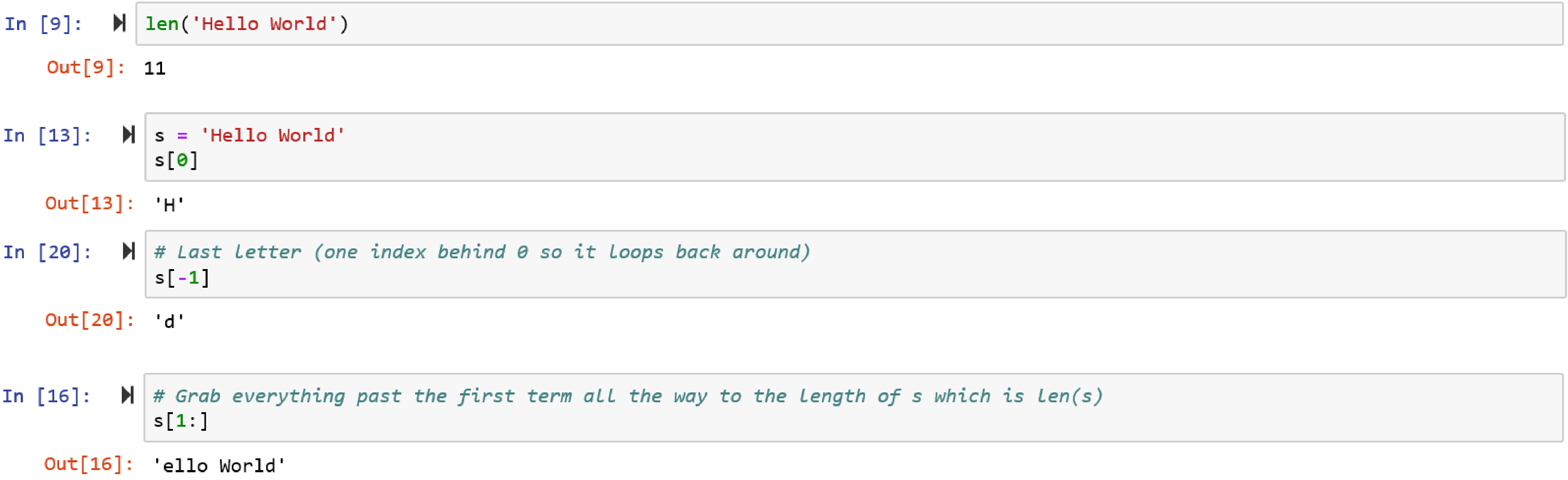
String Slicing: s[x:y:z]

x --start (optional) - Starting integer where the slicing of the object starts. Default to None if not provided.

y -- stop - Integer until which the slicing takes place. The slicing stops at index stop -1 (last element).

Z--step (optional) - Integer value which determines the increment between each index for slicing. Defaults to None if not provided.

- **String length:**



```
In [9]:  len('Hello World')

Out[9]: 11
```

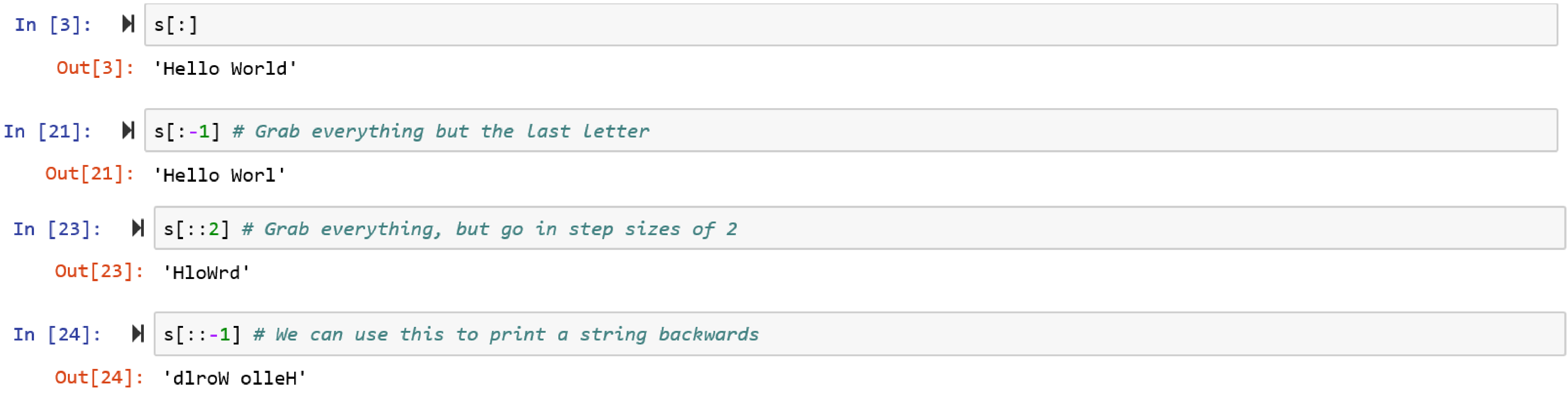
```
In [13]:  s = 'Hello World'
          s[0]

Out[13]: 'H'
```

```
In [20]:  # Last letter (one index behind 0 so it loops back around)
          s[-1]

Out[20]: 'd'
```

```
In [16]:  # Grab everything past the first term all the way to the length of s which is len(s)
          s[1:]

Out[16]: 'ello World'
```

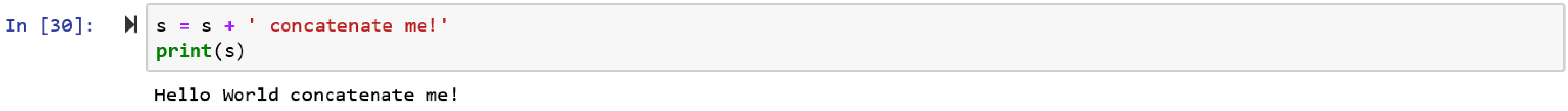
Python Object and Data Structure Basics

Strings



```
In [3]:  s[:]
Out[3]:  'Hello World'

In [21]: s[:-1] # Grab everything but the last letter
Out[21]: 'Hello Worl'

In [23]: s[::2] # Grab everything, but go in step sizes of 2
Out[23]: 'HloWrd'

In [24]: s[::-1] # We can use this to print a string backwards
Out[24]: 'dlroW olleH'
```

- **To Concatenate strings**

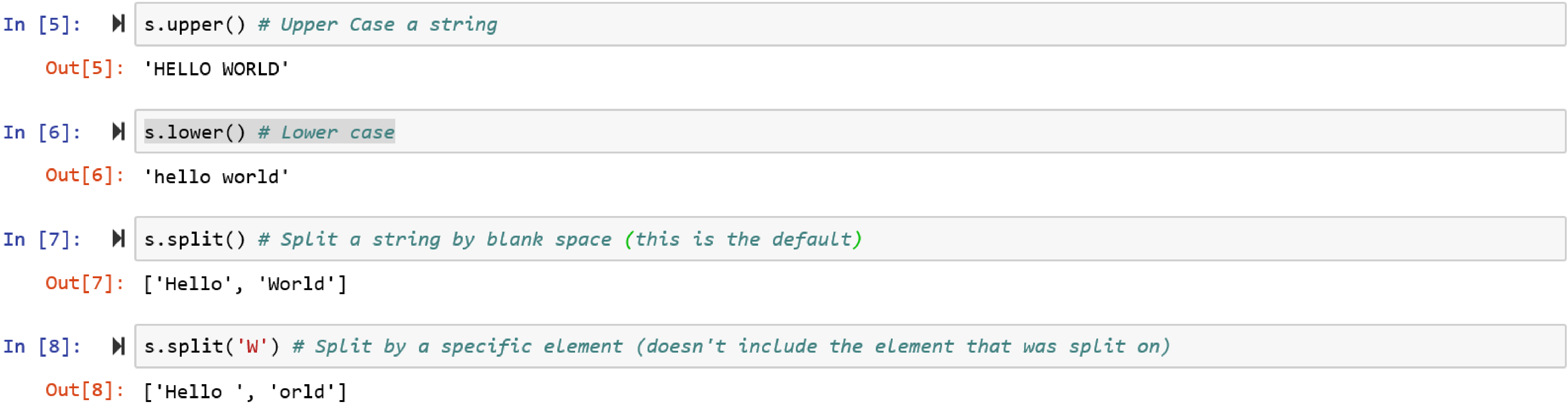


```
In [30]: s = s + ' concatenate me!'
         print(s)

         Hello World concatenate me!
```

Python Object and Data Structure Basics

Strings

- **String Methods**

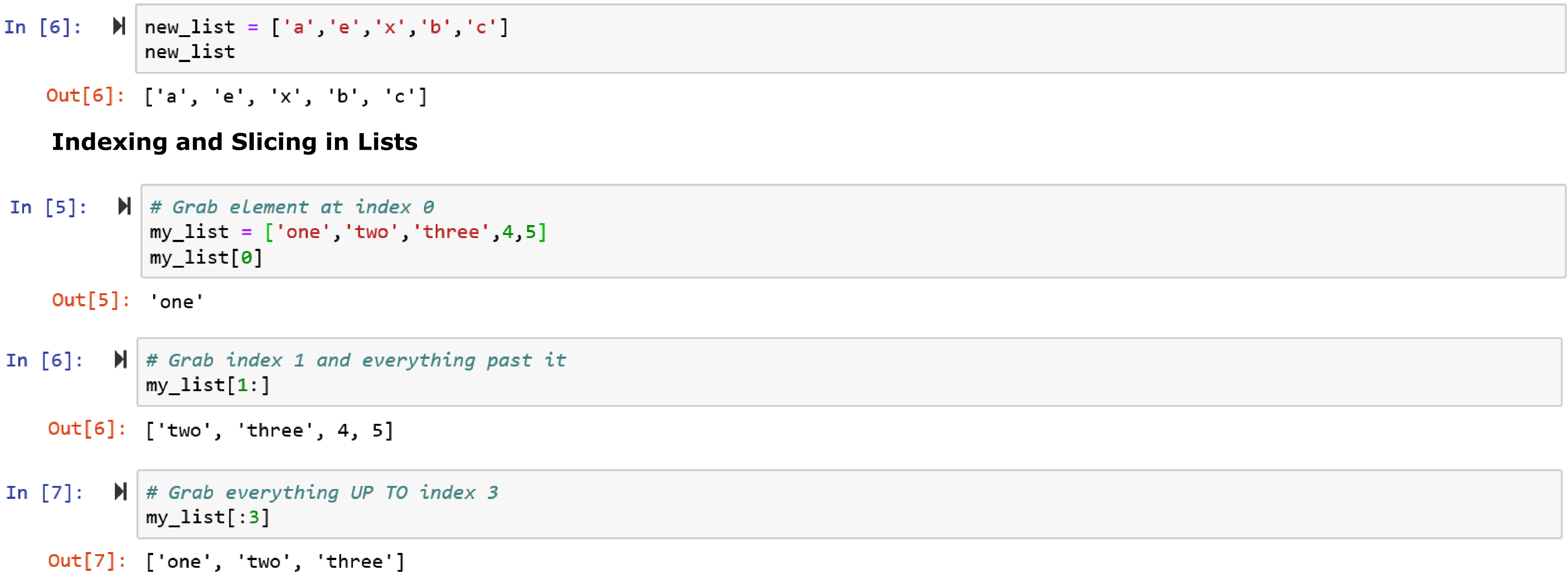


```
In [5]:  s.upper() # Upper Case a string

Out[5]:  'HELLO WORLD'

In [6]:  s.lower() # Lower case

Out[6]:  'hello world'

In [7]:  s.split() # Split a string by blank space (this is the default)

Out[7]:  ['Hello', 'World']

In [8]:  s.split('W') # Split by a specific element (doesn't include the element that was split on)

Out[8]:  ['Hello ', 'orld']
```

Python Object and Data Structure Basics

Lists

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.



```
In [6]:  new_list = ['a','e','x','b','c']
         new_list

Out[6]:  ['a', 'e', 'x', 'b', 'c']
```

Indexing and Slicing in Lists

```
In [5]:  # Grab element at index 0
         my_list = ['one','two','three',4,5]
         my_list[0]

Out[5]:  'one'
```

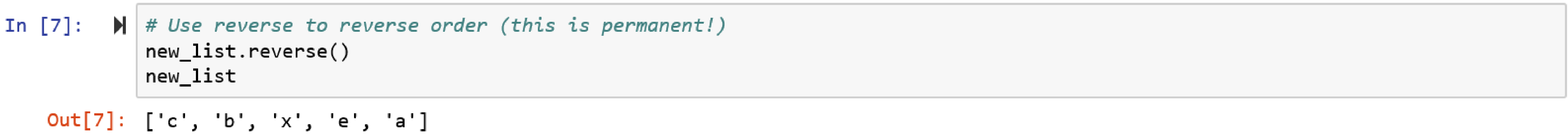
```
In [6]:  # Grab index 1 and everything past it
         my_list[1:]

Out[6]:  ['two', 'three', 4, 5]
```

```
In [7]:  # Grab everything UP TO index 3
         my_list[:3]

Out[7]:  ['one', 'two', 'three']
```

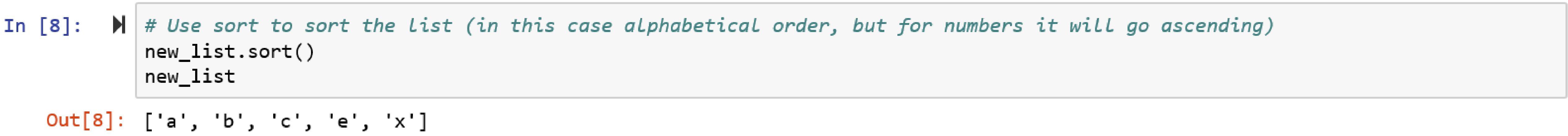
Python Object and Data Structure Basics

Basic List Methods

Append() : Add a single element to end of the list, reverse() : Reverse a List, sort() : sorts elements of a list, pop() : Removes element at given index

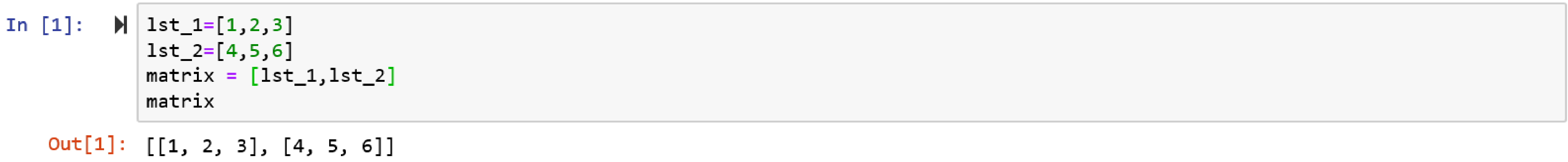


```
In [7]:  # Use reverse to reverse order (this is permanent!)
         new_list.reverse()
         new_list

Out[7]: ['c', 'b', 'x', 'e', 'a']
```



```
In [8]:  # Use sort to sort the list (in this case alphabetical order, but for numbers it will go ascending)
         new_list.sort()
         new_list

Out[8]: ['a', 'b', 'c', 'e', 'x']
```

Nesting Lists

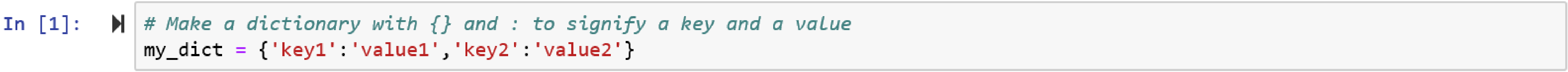


```
In [1]:  lst_1=[1,2,3]
         lst_2=[4,5,6]
         matrix = [lst_1,lst_2]
         matrix

Out[1]: [[1, 2, 3], [4, 5, 6]]
```

Python Object and Data Structure Basics

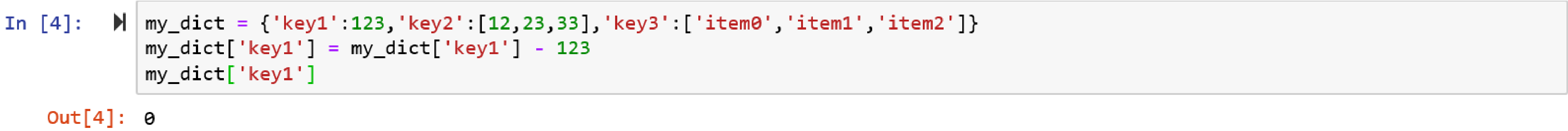
Dictionaries : A **dictionary** is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

Constructing a Dictionary



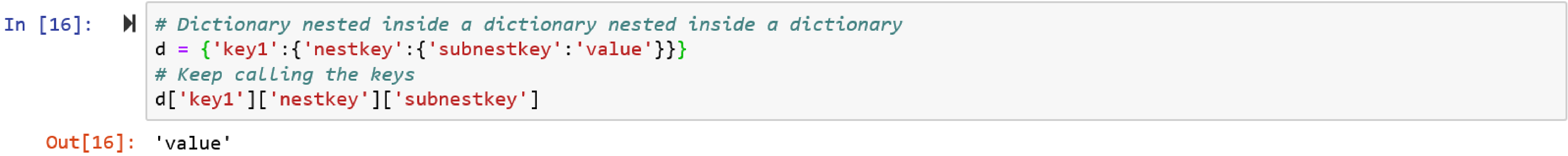
```
In [1]:
# Make a dictionary with {} and : to signify a key and a value
my_dict = {'key1':'value1','key2':'value2'}
```

Accessing objects from a dictionary



```
In [4]:
my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}
my_dict['key1'] = my_dict['key1'] - 123
my_dict['key1']

Out[4]: 0
```

Nesting Dictionaries

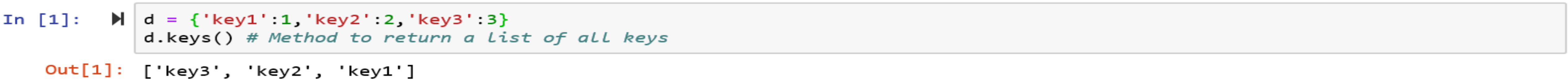


```
In [16]:
# Dictionary nested inside a dictionary nested inside a dictionary
d = {'key1':{'nestkey':{'subnestkey':'value'}}}
# Keep calling the keys
d['key1']['nestkey']['subnestkey']

Out[16]: 'value'
```

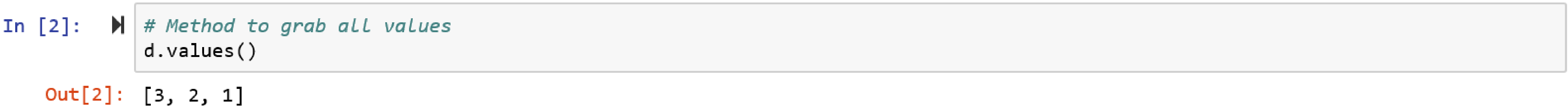
Python Object and Data Structure Basics

Dictionaries **Basic Dictionary Methods**

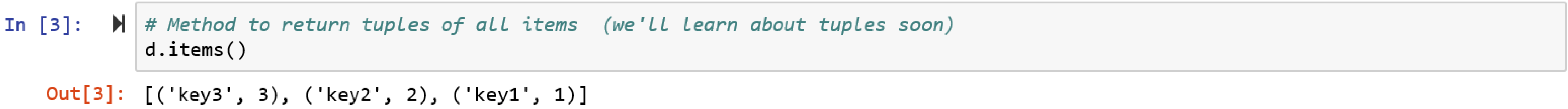
Values() : Return a list of all values in the dictionary, keys() : Returns a list containing the dictionary’s keys, items() : Return a list containing a tuple for each key value pair.



```
In [1]:  d = {'key1':1,'key2':2,'key3':3}
         d.keys() # Method to return a list of all keys

Out[1]:  ['key3', 'key2', 'key1']
```



```
In [2]:  # Method to grab all values
         d.values()

Out[2]:  [3, 2, 1]
```

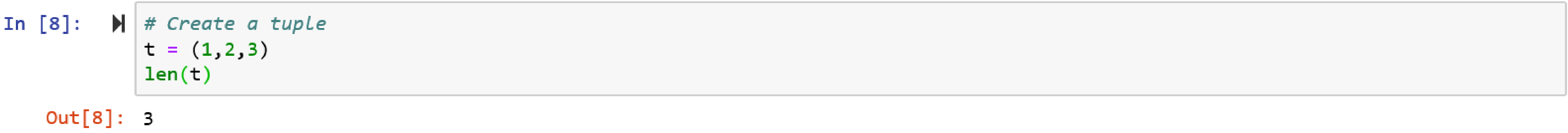


```
In [3]:  # Method to return tuples of all items  (we'll learn about tuples soon)
         d.items()

Out[3]:  [('key3', 3), ('key2', 2), ('key1', 1)]
```

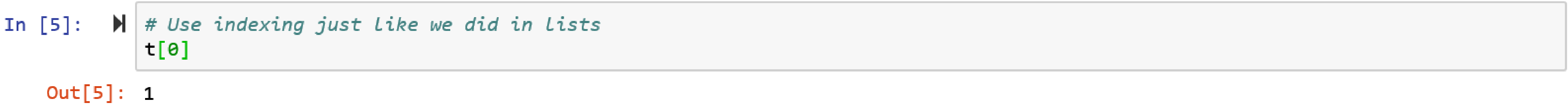
Python Object and Data Structure Basics

Tuples : A **tuple** is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

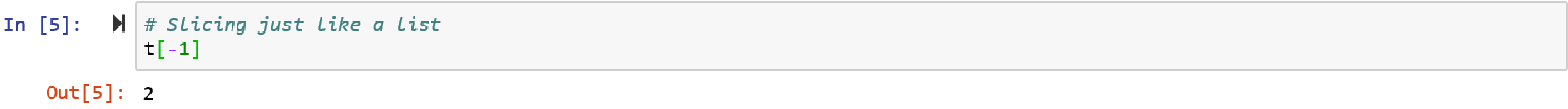
- Constructing Tuples:



```
In [8]:  # Create a tuple
         t = (1,2,3)
         len(t)

Out[8]:  3
```



```
In [5]:  # Use indexing just like we did in lists
         t[0]

Out[5]:  1
```



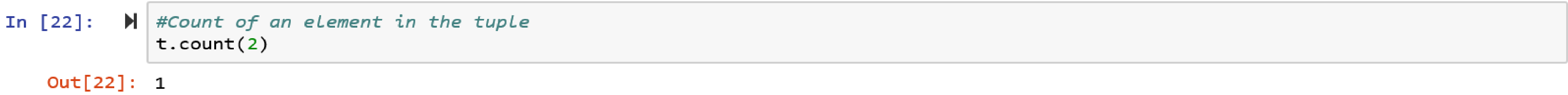
```
In [5]:  # Slicing just like a list
         t[-1]

Out[5]:  2
```

Python Object and Data Structure Basics

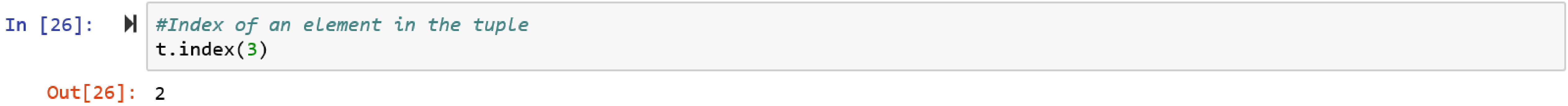
Tuples **Basic Tuple Methods**

count() returns occurrences of element in a tuple

index() returns smallest index of element in tuple

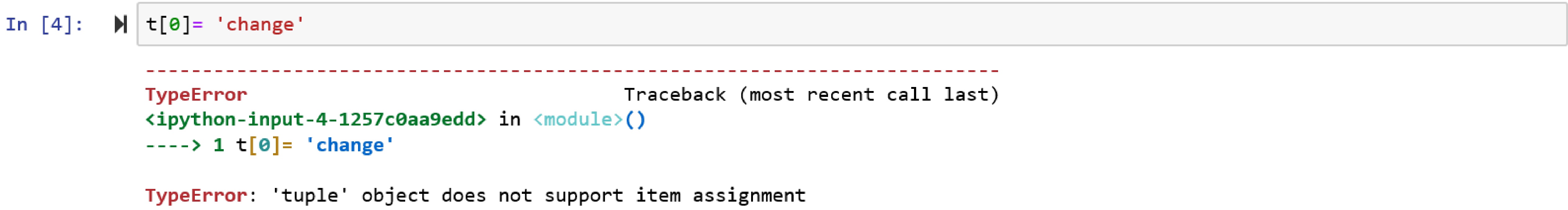


```
In [22]:  #Count of an element in the tuple
          t.count(2)

Out[22]:  1
```



```
In [26]:  #Index of an element in the tuple
          t.index(3)

Out[26]:  2
```

- Tuples are immutable.



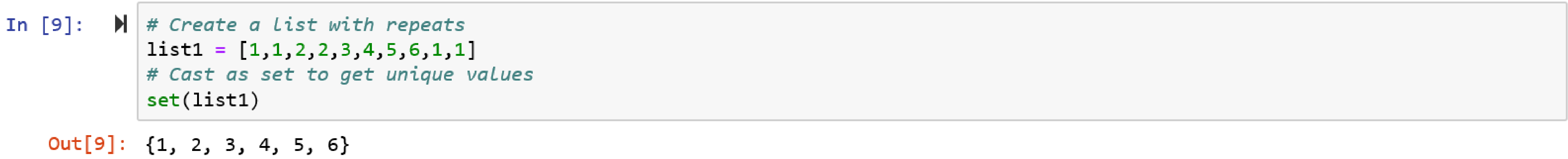
```
In [4]:  t[0]= 'change'

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-1257c0aa9edd> in <module>()
----> 1 t[0]= 'change'

TypeError: 'tuple' object does not support item assignment
```

Python Object and Data Structure Basics

Sets and Booleans

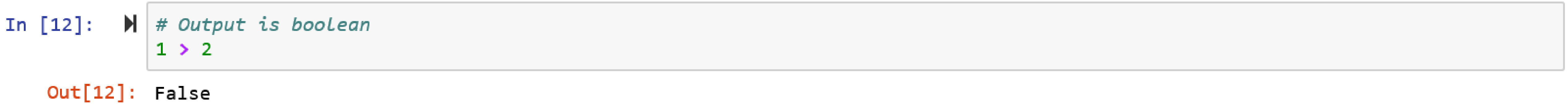
Sets: Sets are an unordered collection of *unique* elements. We can construct them by using the set() function.

- Note the curly brackets. This does not indicate a dictionary! Although you can draw analogies as a set being a dictionary with only keys.

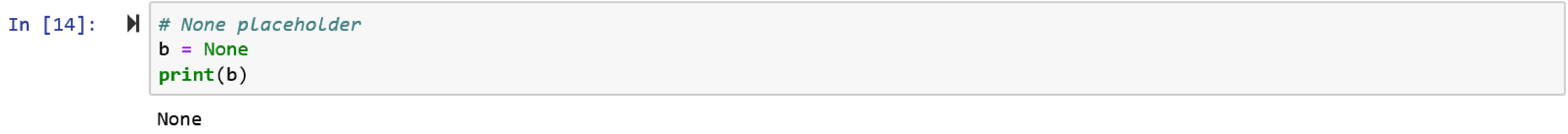


```
In [9]:   # Create a list with repeats
          list1 = [1,1,2,2,3,4,5,6,1,1]
          # Cast as set to get unique values
          set(list1)

Out[9]:   {1, 2, 3, 4, 5, 6}
```

Boolean: It also has a placeholder object called None. (for an object that we don't want to reassign yet)



```
In [12]:  # Output is boolean
          1 > 2

Out[12]:  False
```

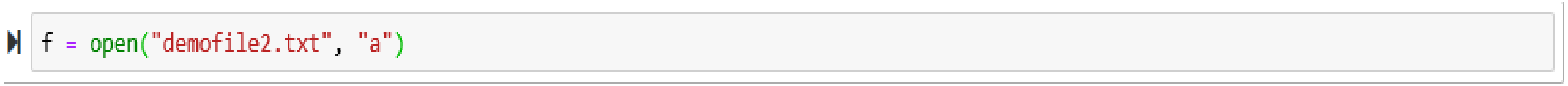


```
In [14]:  # None placeholder
          b = None
          print(b)

          None
```

Python Object and Data Structure Basics

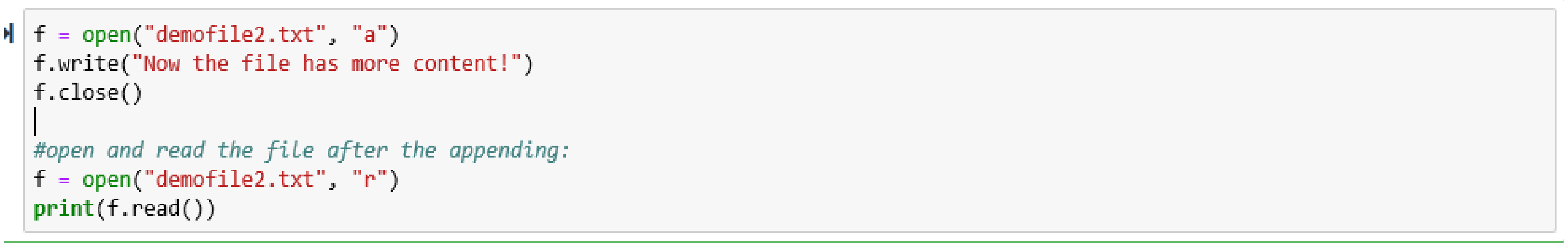
Files

- Python uses file objects to interact with external files on your computer. Various functions we can perform on files are:
  - **Opening a file**



```
f = open("demofile2.txt", "a")
```

  - **Writing to a file**



```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

To write to an existing file, you must add a parameter to the **open()** function:

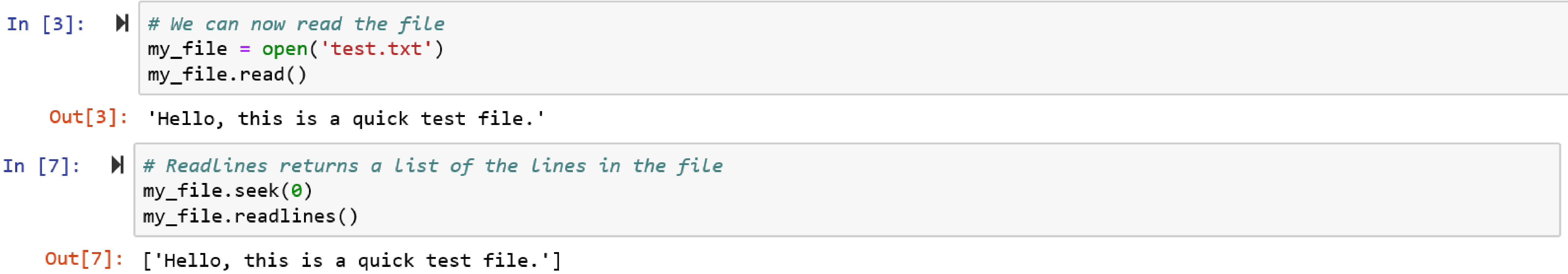
"a" - **Append** - will append to the end of the file

"w" - **Write** - will overwrite any existing content

Python Object and Data Structure Basics

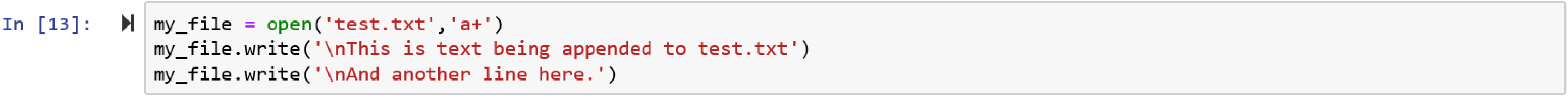
Files

– Reading from a file



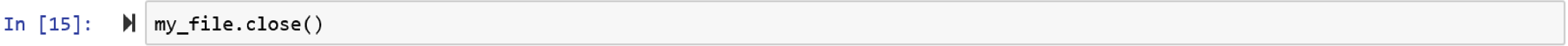
```
In [3]:  # We can now read the file
         my_file = open('test.txt')
         my_file.read()

Out[3]:  'Hello, this is a quick test file.'

In [7]:  # Readlines returns a list of the lines in the file
         my_file.seek(0)
         my_file.readlines()

Out[7]:  ['Hello, this is a quick test file.']
```

– Appending to a file



```
In [13]: my_file = open('test.txt','a+')
         my_file.write('\nThis is text being appended to test.txt')
         my_file.write('\nAnd another line here.')
```

– Closing a file



```
In [15]: my_file.close()
```

Contents

Python object and data structures Basics

Python Comparison Operators

Python Statements

Methods and Functions

Modules and Packages

Errors and Exception Handling

Built-in Functions

Python Comparison Operators

Comparison Operators

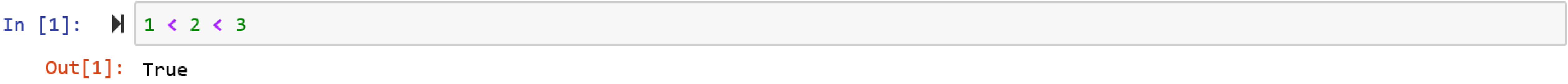
- Comparison Operators in Python will allow us to compare variables and output a Boolean value (True or False).
- Below are the various operators which are commonly used.

Operator	Symbol	Description	Example
Equal	==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
Not Equal	!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true
Greater than	>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
Less than	<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
Greater than or Equal	>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
Less than or Equal	<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Comparison Operators

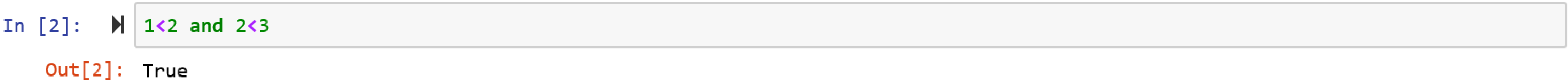
Chained Comparison Operators

- **Chained Expressions:**



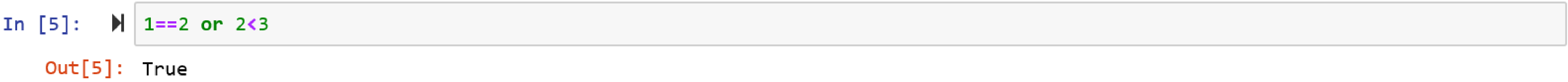
```
In [1]:  1 < 2 < 3
Out[1]: True
```

- **AND Operator:**



```
In [2]:  1<2 and 2<3
Out[2]: True
```

- **OR Operator:**



```
In [5]:  1==2 or 2<3
Out[5]: True
```

Contents

Python object and data structures Basics

Python Comparison Operators

Python Statements

Methods and Functions

Modules and Packages

Errors and Exception Handling

Built-in Functions

Python Statements

Introduction to Python Statements

- There are two reasons we take this approach for learning the context of Python Statements:
  - If you are coming from a different language this will rapidly accelerate your understanding of Python.
  - Learning about statements will allow you to be able to read other languages more easily in the future.

Python Vs Other Languages:

```
if x:
    if y:
        code-statement
else:
    another-code-statement
```

- Python is so heavily driven by code indentation and whitespace.
- Code readability is a core part of the design of the Python language.
- Python gets rid of () and {} by incorporating two main factors: a *colon* and *whitespace*.
- Another major difference is the lack of semicolons in Python

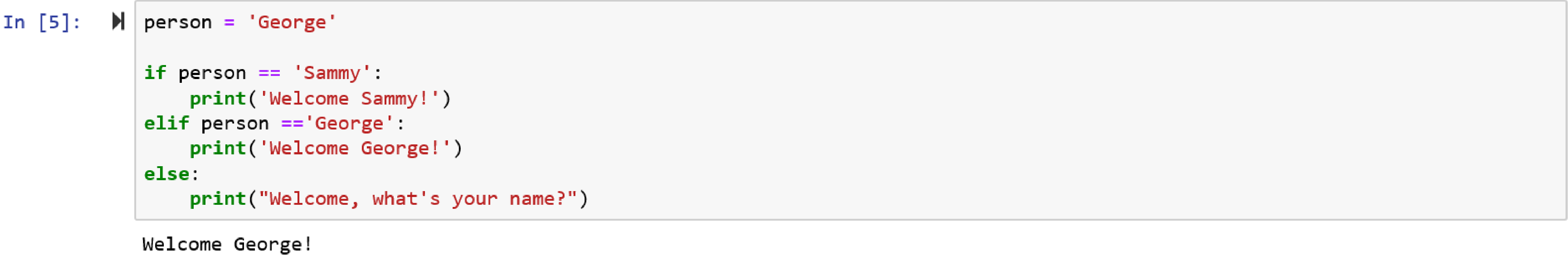
Python Statements

if, elif, else Statements

- Pseudo Code:

```
if case1:
    perform action1
elif case2:
    perform action2
else:
    perform action3
```

- Example:



```python
person = 'George'

if person == 'Sammy':
    print('Welcome Sammy!')
elif person =='George':
    print('Welcome George!')
else:
    print("Welcome, what's your name?")
```

```
Welcome George!
```

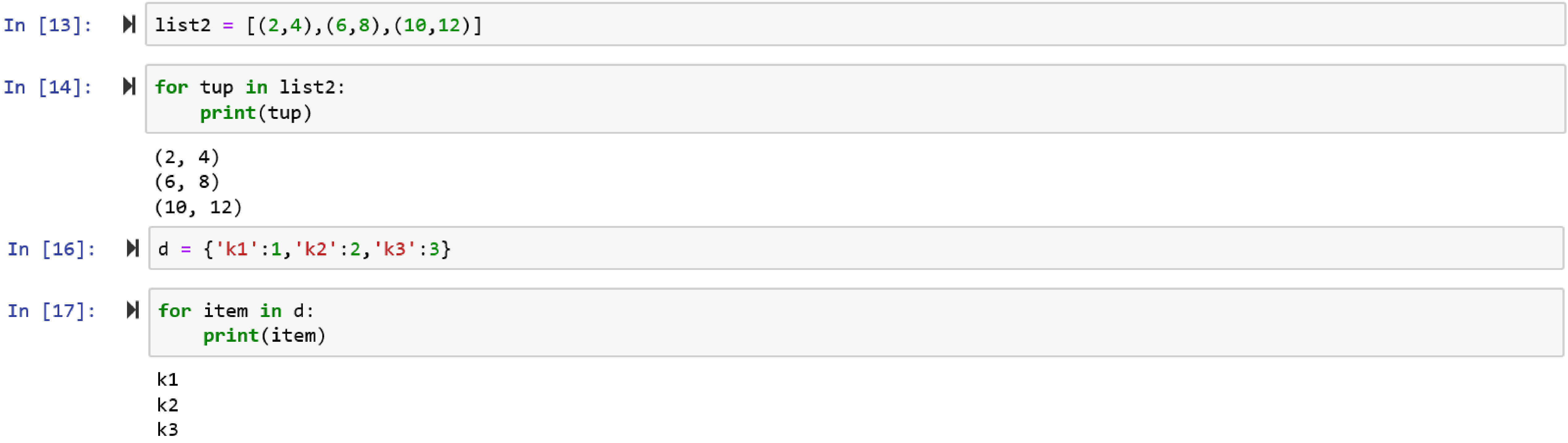
Python Statements

For Loop, While Loop

- FOR Loop Pseudo code:

```
for item in object:
    statements to do stuff
```

- Examples:



```
In [13]:  list2 = [(2,4),(6,8),(10,12)]

In [14]:  for tup in list2:
              print(tup)

          (2, 4)
          (6, 8)
          (10, 12)

In [16]:  d = {'k1':1,'k2':2,'k3':3}

In [17]:  for item in d:
              print(item)

          k1
          k2
          k3
```

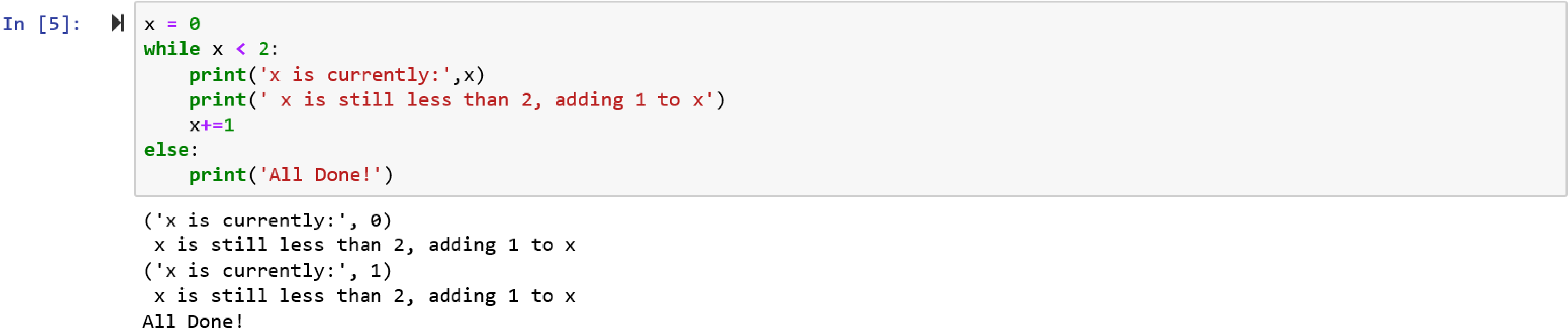
Python Statements

For Loop, While Loop

- WHILE Loop Pseudo Code:

```
while test:
    code statements
else:
    final code statements
```

- Examples:



In [5]:

```
x = 0
while x < 2:
    print('x is currently:',x)
    print(' x is still less than 2, adding 1 to x')
    x+=1
else:
    print('All Done!')
```

```
('x is currently:', 0)
 x is still less than 2, adding 1 to x
('x is currently:', 1)
 x is still less than 2, adding 1 to x
All Done!
```

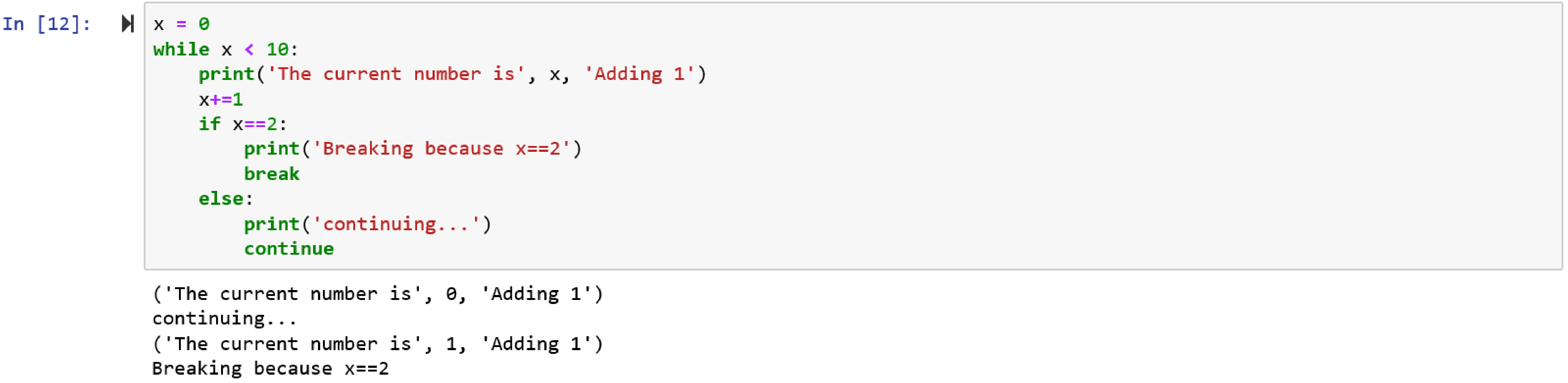
Python Statements

break, continue, pass

- Pseudo Code:

```
while test:
   code statement
if test:
   break        /*Breaks out of the current closest enclosing loop*/
if test:
   continue     /*Goes to the top of the closest enclosing loop*/
else:
```

- Example:



In [12]:

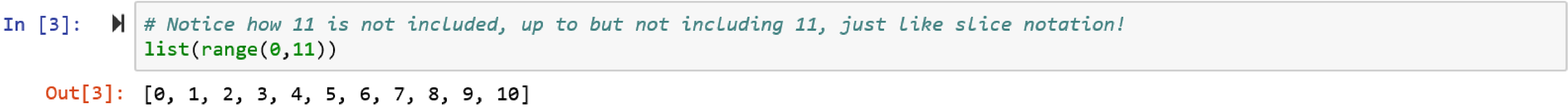
```
x = 0
while x < 10:
    print('The current number is', x, 'Adding 1')
    x+=1
    if x==2:
        print('Breaking because x==2')
        break
    else:
        print('continuing...')
        continue
```

```
('The current number is', 0, 'Adding 1')
continuing...
('The current number is', 1, 'Adding 1')
Breaking because x==2
```

Python Statements

Useful Operators

- **range():** The range function allows you to quickly generate a list of integers.



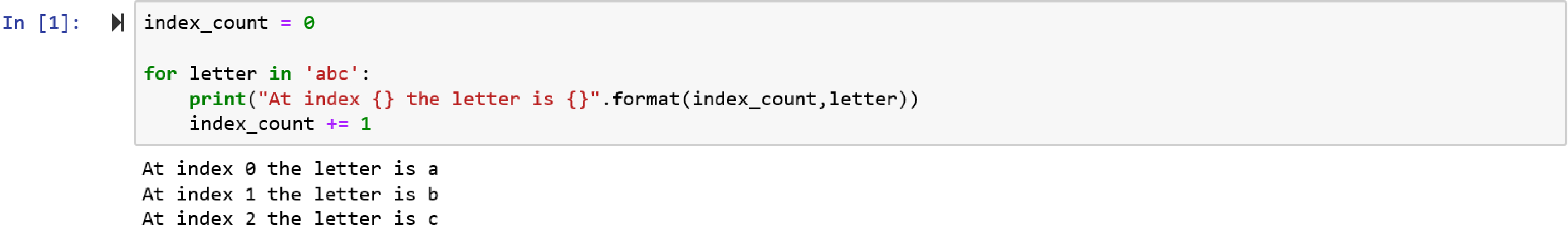
In [3]:

```
# Notice how 11 is not included, up to but not including 11, just like slice notation!
list(range(0,11))
```

Out[3]:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- **Enumerate():** Enumerate is a very useful function to use with for loops. It keeps track of how many loops are done and automatically creates and updates the index_count or loop_count variable.



In [1]:

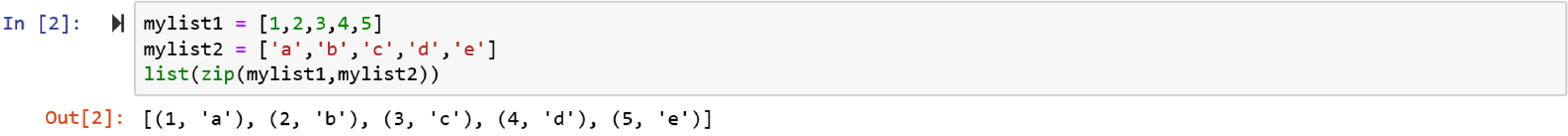
```
index_count = 0

for letter in 'abc':
    print("At index {} the letter is {}".format(index_count,letter))
    index_count += 1
```

```
At index 0 the letter is a
At index 1 the letter is b
At index 2 the letter is c
```

Python Statements

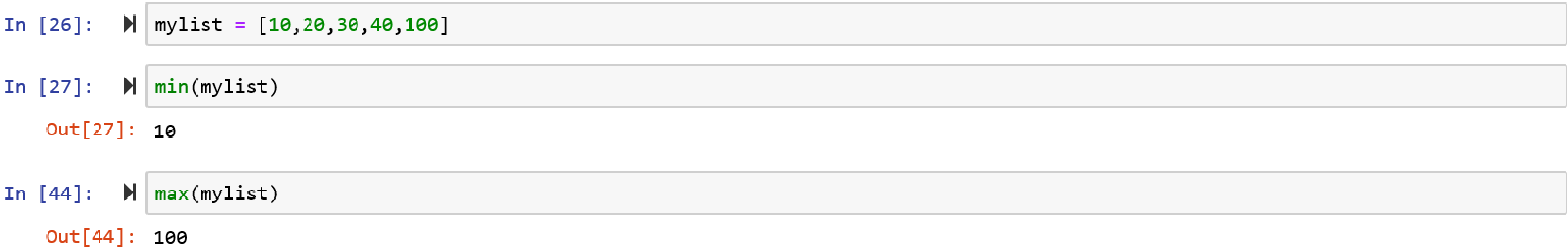
Useful Operators

- **Zip():** You can use the zip() function to quickly create a list of tuples by "zipping" up together two lists.



```
In [2]:   mylist1 = [1,2,3,4,5]
          mylist2 = ['a','b','c','d','e']
          list(zip(mylist1,mylist2))

Out[2]:  [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

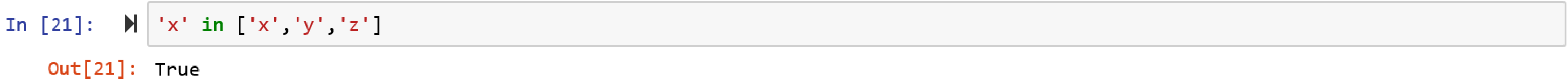
- **Min() and max():** We can check the minimum or maximum of a list with these functions.



```
In [26]:   mylist = [10,20,30,40,100]

In [27]:   min(mylist)

Out[27]:  10

In [44]:   max(mylist)

Out[44]:  100
```

Python Statements
Useful Operators

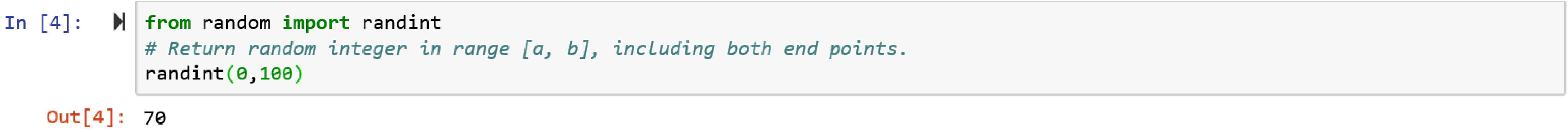
- **In():** Keyword usually used in loops. Can also used to check if an object is in a list.



```
In [21]:  'x' in ['x','y','z']
Out[21]:  True
```

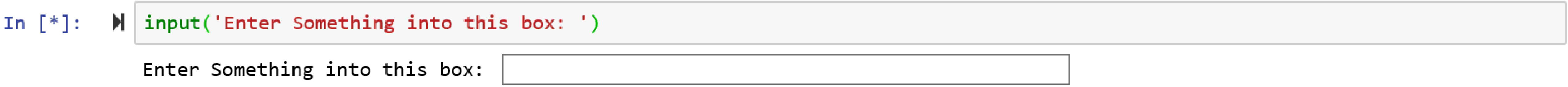
- **Random():** Python comes with a built in random library. There are a lot of functions included in this random library

  Example: randint(min_range_value, max_range_value)



```
In [4]:  from random import randint
         # Return random integer in range [a, b], including both end points.
         randint(0,100)
Out[4]:  70
```

- **Input()**

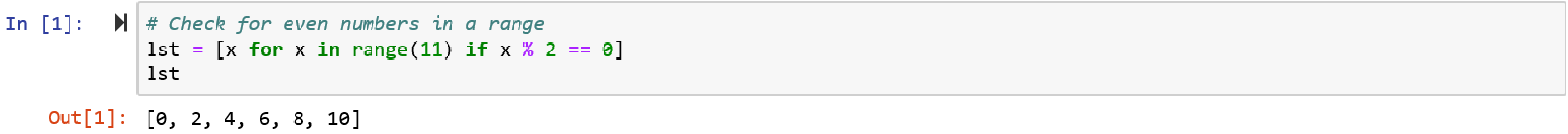


```
In [*]:  input('Enter Something into this box: ')
         Enter Something into this box:
```

Python Statements

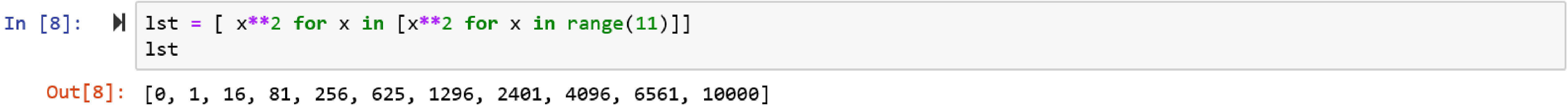
List Comprehensions

- List comprehensions allow us to build out lists using a different notation. It is essentially a one line for loop built inside of brackets.
- **Example:**



```
In [1]:    # Check for even numbers in a range
           lst = [x for x in range(11) if x % 2 == 0]
           lst

Out[1]:    [0, 2, 4, 6, 8, 10]
```

- We can also perform nested list comprehensions.



```
In [8]:    lst = [ x**2 for x in [x**2 for x in range(11)]]
           lst

Out[8]:    [0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000]
```

Contents

Python object and data structures Basics

Python Comparison Operators

Python Statements

Methods and Functions

Modules and Packages

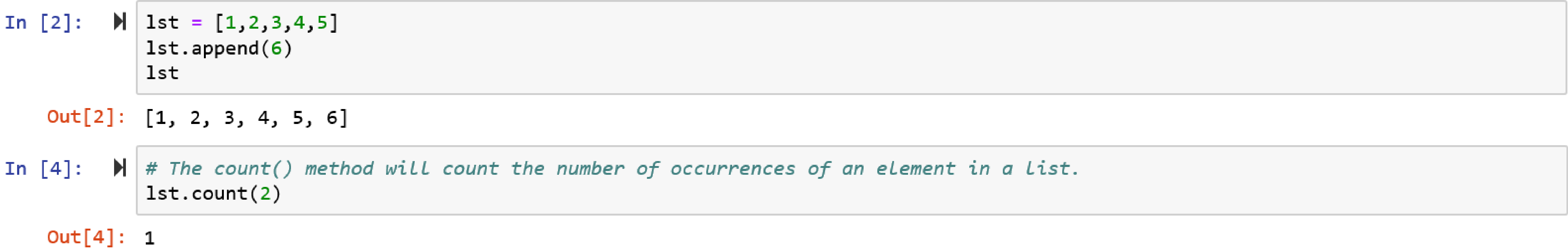
Errors and Exception Handling

Built-in Functions

Methods and Functions

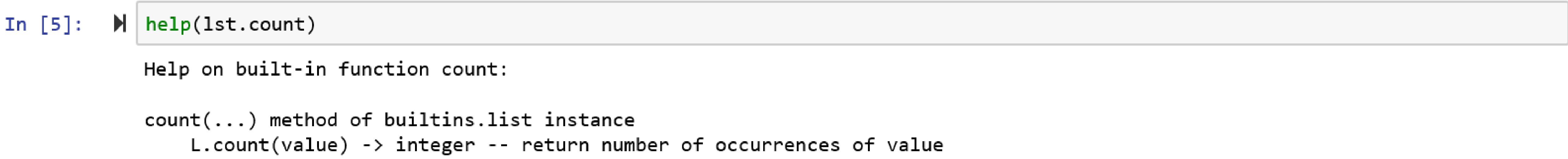
Methods

- Methods are essentially functions built into objects.
- The general syntax for methods is, **object. Method(arg1,arg2,etc…)**
- Examples of methods for a list - Append(), Count(), Extend(), Insert(), Pop(), Remove(), Reverse(), Sort()



```
In [2]:  lst = [1,2,3,4,5]
         lst.append(6)
         lst

Out[2]:  [1, 2, 3, 4, 5, 6]

In [4]:  # The count() method will count the number of occurrences of an element in a list.
         lst.count(2)

Out[4]:  1
```

- Help() function is used to get more information about the method.



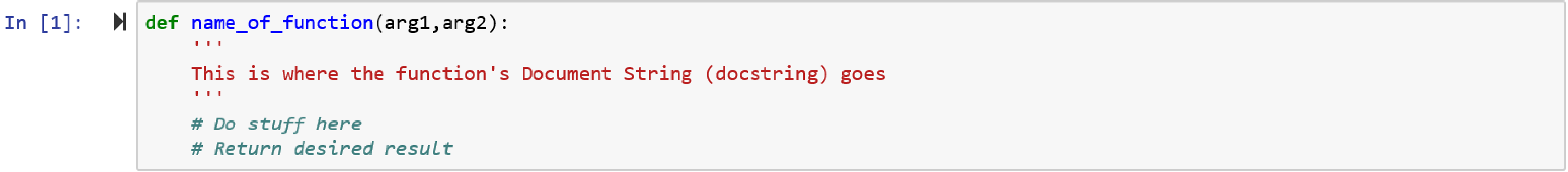
```
In [5]:  help(lst.count)

         Help on built-in function count:

         count(...) method of builtins.list instance
             L.count(value) -> integer -- return number of occurrences of value
```

Methods and Functions

Functions

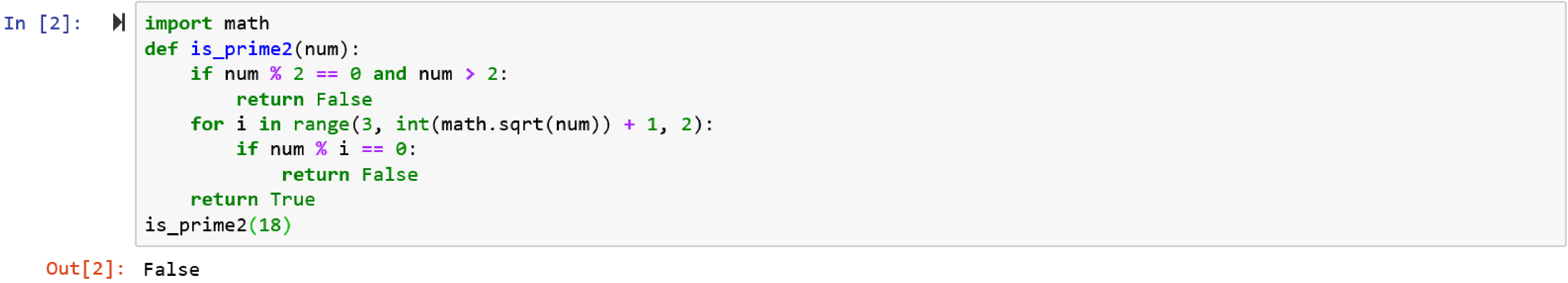
- Functions are one of most basic levels of reusing code in Python.

Def Statements: Below is the syntax.



```
def name_of_function(arg1,arg2):
    '''
    This is where the function's Document String (docstring) goes
    '''
    # Do stuff here
    # Return desired result
```

Return(): Allows a function to return a result that can then be stored as a variable.



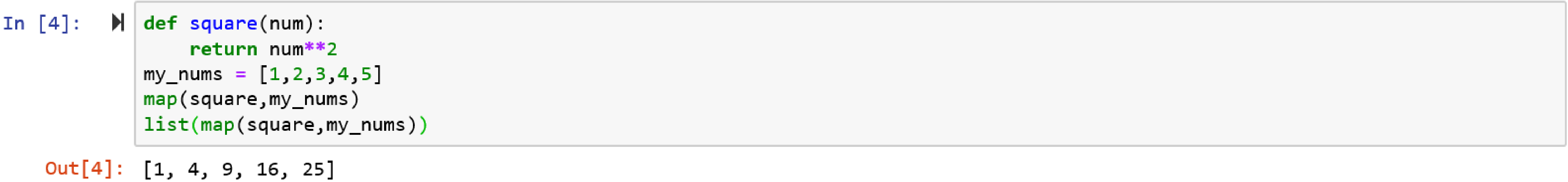
```
import math
def is_prime2(num):
    if num % 2 == 0 and num > 2:
        return False
    for i in range(3, int(math.sqrt(num)) + 1, 2):
        if num % i == 0:
            return False
    return True
is_prime2(18)
```

```
Out[2]: False
```

Methods and Functions

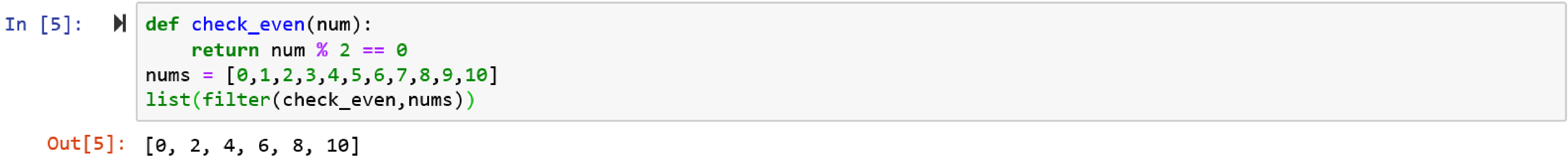
Lambda Expressions, map and filter

- **MAP():** Allows you to map a function to an iterable object.
- Syntax: **map(<function_name>,<function arguments>)**



```
In [4]:   def square(num):
              return num**2
          my_nums = [1,2,3,4,5]
          map(square,my_nums)
          list(map(square,my_nums))

Out[4]:   [1, 4, 9, 16, 25]
```

- **FILTER():** Can be used with a function that returns either True or False.
- Syntax: **filter(<function_name>,<function arguments>)**

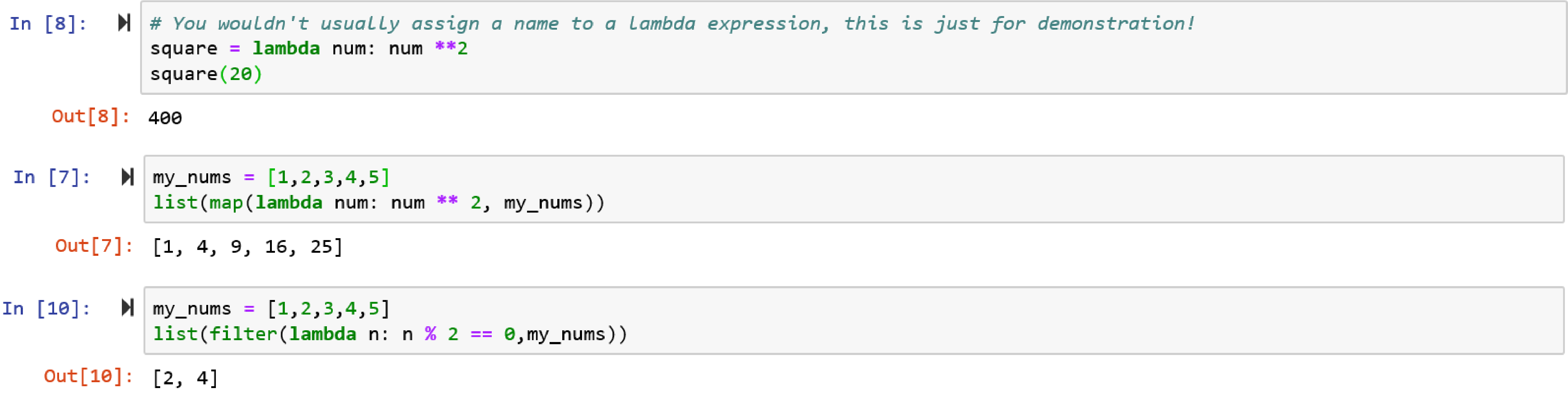


```
In [5]:   def check_even(num):
              return num % 2 == 0
          nums = [0,1,2,3,4,5,6,7,8,9,10]
          list(filter(check_even,nums))

Out[5]:   [0, 2, 4, 6, 8, 10]
```

Methods and Functions

Lambda Expressions, map and filter

- **Lambda Expression:** Lambda expressions allow us to create "anonymous" functions. We can create ad-hoc functions without ‘def’
- Lambda is designed for coding simple functions, and def handles the larger tasks.
- You will find yourself using lambda expressions often with certain non-built-in libraries, for example the pandas library for data analysis works very well with lambda expressions.

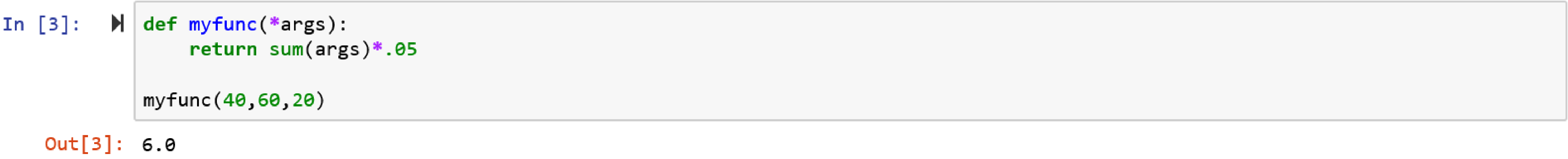


```
In [8]:  # You wouldn't usually assign a name to a lambda expression, this is just for demonstration!
          square = lambda num: num **2
          square(20)

Out[8]:   400

In [7]:   my_nums = [1,2,3,4,5]
          list(map(lambda num: num ** 2, my_nums))

Out[7]:   [1, 4, 9, 16, 25]

In [10]:  my_nums = [1,2,3,4,5]
          list(filter(lambda n: n % 2 == 0,my_nums))

Out[10]:  [2, 4]
```

Methods and Functions

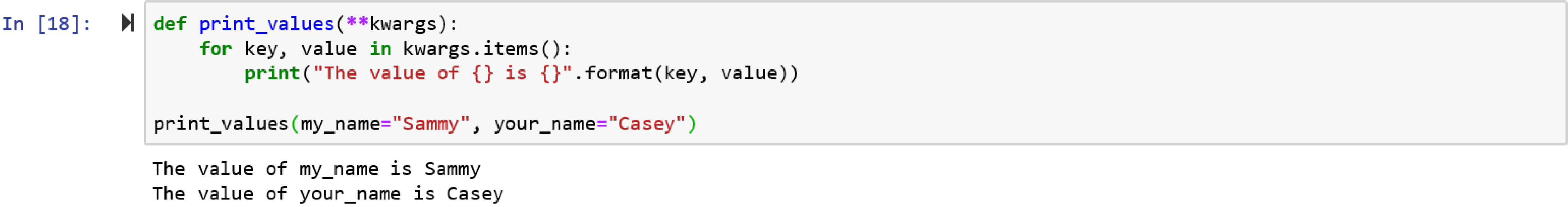
*args and **kwargs

- ***args:** It allows an arbitrary number of arguments, and the function takes them in as a tuple of values. Syntax: **def myfunc(*args):**



```
In [3]:  def myfunc(*args):
             return sum(args)*.05

         myfunc(40,60,20)

Out[3]:  6.0
```

- ****kwargs:** Builds a dictionary of key/value pairs. Syntax - **def myfunc(**kwargs):**



```
In [18]:  def print_values(**kwargs):
              for key, value in kwargs.items():
                  print("The value of {} is {}".format(key, value))

          print_values(my_name="Sammy", your_name="Casey")

          The value of my_name is Sammy
          The value of your_name is Casey
```

- We can pass *args and **kwargs into the same function, but *args have to appear before **kwargs

Syntax: **def myfunc(*args, **kwargs):**

Contents

Python object and data structures Basics

Python Comparison Operators

Python Statements

Methods and Functions

Modules and Packages

Errors and Exception Handling

Built-in Functions

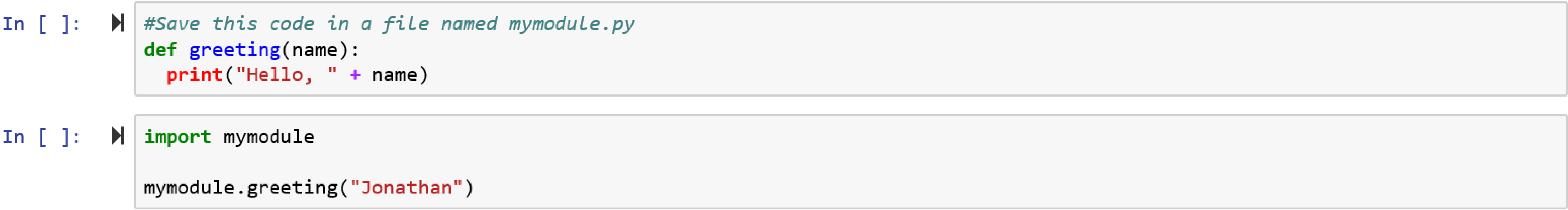
Modules and Packages

Modules, built-in modules and writing modules

Modules: Modules in Python are simply Python files with the .py extension, which implement a set of functions.

dir() – looks for functions implemented in each module

Writing Modules: To create a module of your own, simply create a new .py file with the module name, and then import it using the Python file name (without the .py extension) using the import command.



In [ ]:

```
#Save this code in a file named mymodule.py
def greeting(name):
  print("Hello, " + name)
```

In [ ]:

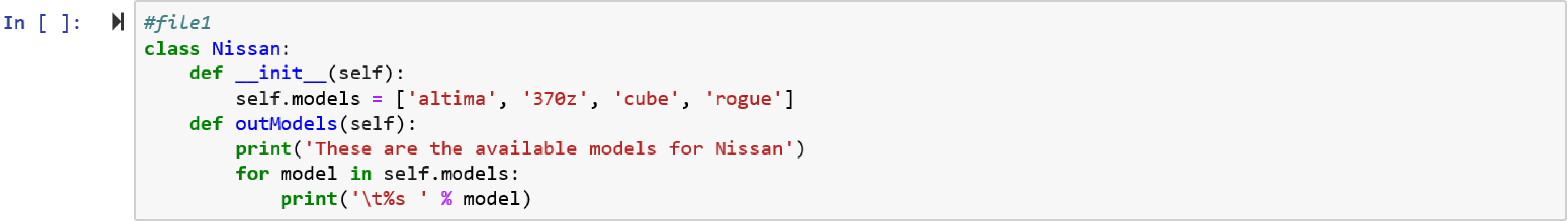
```
import mymodule

mymodule.greeting("Jonathan")
```

Modules and Packages

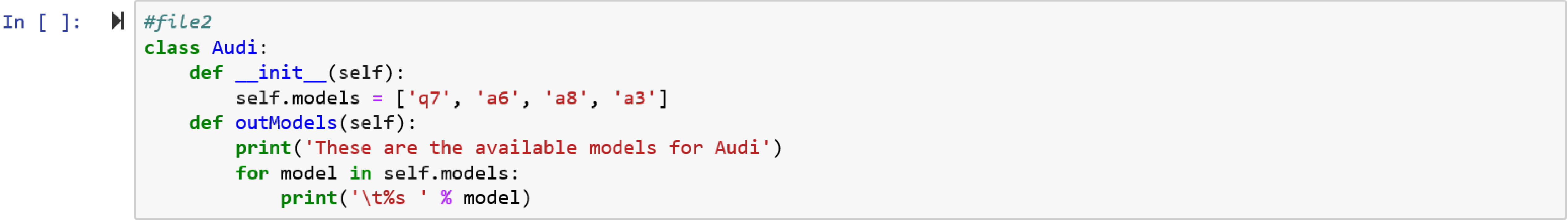
Writing Packages

Writing Packages: Each package in Python is a directory which MUST contain a special file called **_init_.py**.

- The **_init_.py** file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the **_all_** variable



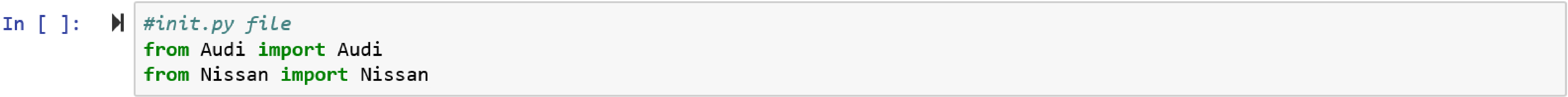
```
In [ ]:  #file1
         class Nissan:
             def __init__(self):
                 self.models = ['altima', '370z', 'cube', 'rogue']
             def outModels(self):
                 print('These are the available models for Nissan')
                 for model in self.models:
                     print('\t%s ' % model)
```



```
In [ ]:  #file2
         class Audi:
             def __init__(self):
                 self.models = ['q7', 'a6', 'a8', 'a3']
             def outModels(self):
                 print('These are the available models for Audi')
                 for model in self.models:
                     print('\t%s ' % model)
```

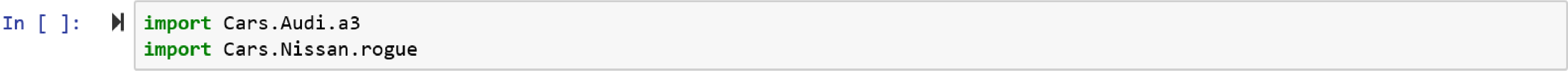
Modules and Packages

Writing Packages

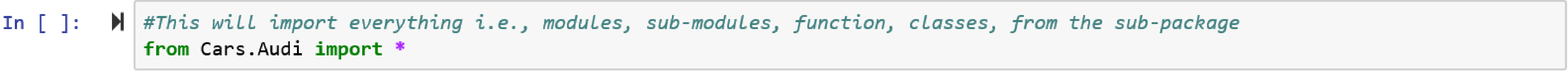


```
#init.py file
from Audi import Audi
from Nissan import Nissan
```

- To import the packages and modules,

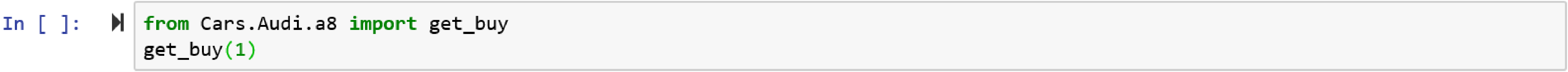


```
import Cars.Audi.a3
import Cars.Nissan.rogue
```



```
#This will import everything i.e., modules, sub-modules, function, classes, from the sub-package
from Cars.Audi import *
```

- For example, Audi’s module a8 has a function get_buy(), we can import it as follows.



```
from Cars.Audi.a8 import get_buy
get_buy(1)
```

Contents

Python object and data structures Basics

Python Comparison Operators

Python Statements

Methods and Functions

Modules and Packages

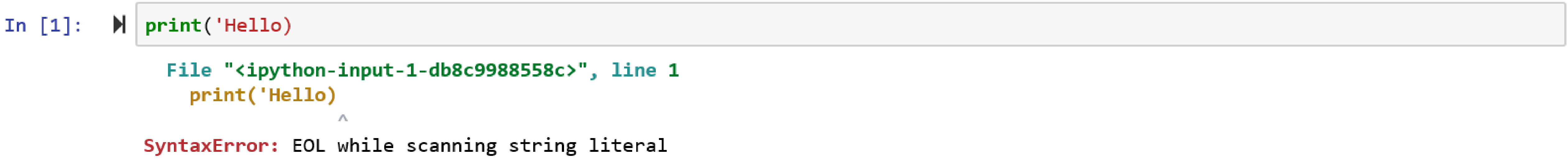
Errors and Exception Handling

Built-in Functions

Errors and Exception Handling

Errors, Exceptions

Error:



```
In [1]:  print('Hello)

  File "<ipython-input-1-db8c9988558c>", line 1
    print('Hello)
                 ^
SyntaxError: EOL while scanning string literal
```

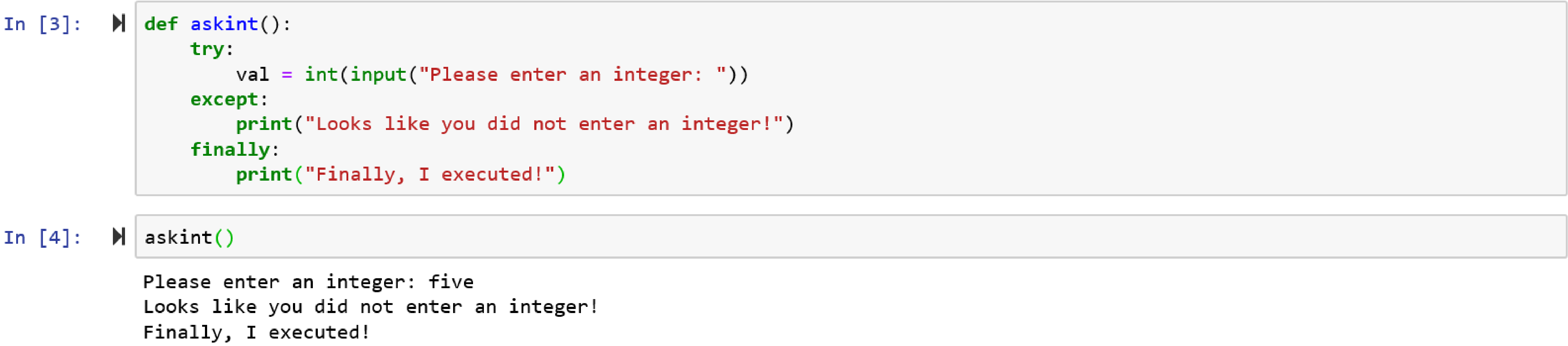
Exception: The type of error and description is known as an Exception.

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
- Errors detected during execution are called exceptions and are not unconditionally fatal.
- You can check out the full list of built-in exceptions here.

Errors and Exception Handling

Exception Handling – try, except, finally

- **try:** The code which can cause an exception to occur is put in the try block
- **except:** The handling of the exception is then implemented in the except block of code.
- **finally:** The finally: block of code will always be run regardless if there was an exception in the try code block. The syntax is:



```
In [3]:  def askint():
             try:
                 val = int(input("Please enter an integer: "))
             except:
                 print("Looks like you did not enter an integer!")
             finally:
                 print("Finally, I executed!")
```

```
In [4]:  askint()
```

```
Please enter an integer: five
Looks like you did not enter an integer!
Finally, I executed!
```

Contents

Python object and data structures Basics

Python Comparison Operators

Python Statements

Methods and Functions

Modules and Packages

Errors and Exception Handling

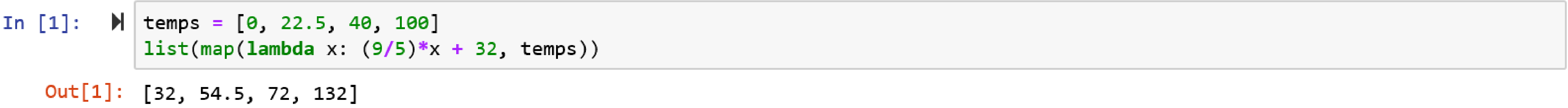
Built-in Functions

Python Built-in Functions

Map()

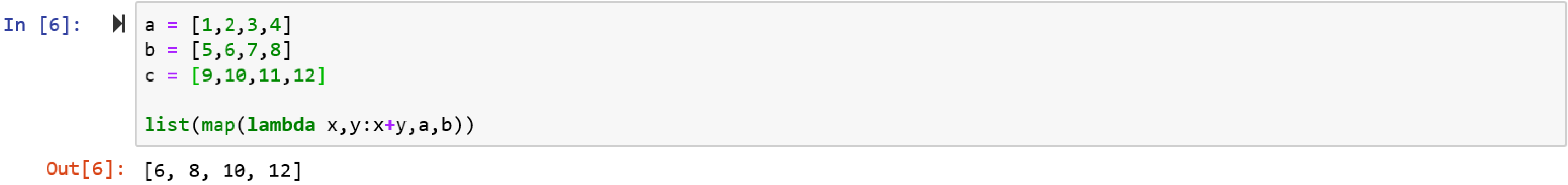
map(): A built-in Python function that takes in two or more arguments: a function and one or more iterables, in the form:

map(function, iterable, ...)



```
In [1]:  temps = [0, 22.5, 40, 100]
         list(map(lambda x: (9/5)*x + 32, temps))

Out[1]:  [32, 54.5, 72, 132]
```

map() with multiple iterables: The iterables should be the same length. In the event that they are not, map() will stop as soon as the shortest iterable is exhausted.



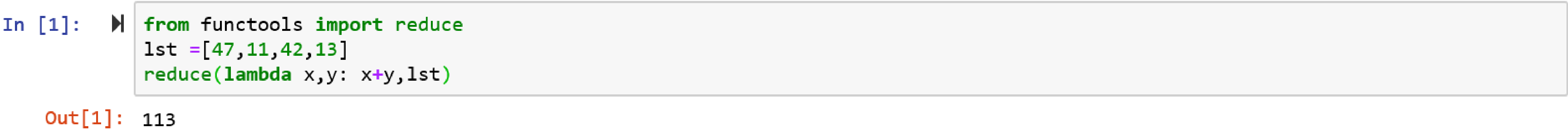
```
In [6]:  a = [1,2,3,4]
         b = [5,6,7,8]
         c = [9,10,11,12]

         list(map(lambda x,y:x+y,a,b))

Out[6]:  [6, 8, 10, 12]
```

Python Built-in Functions

Reduce()

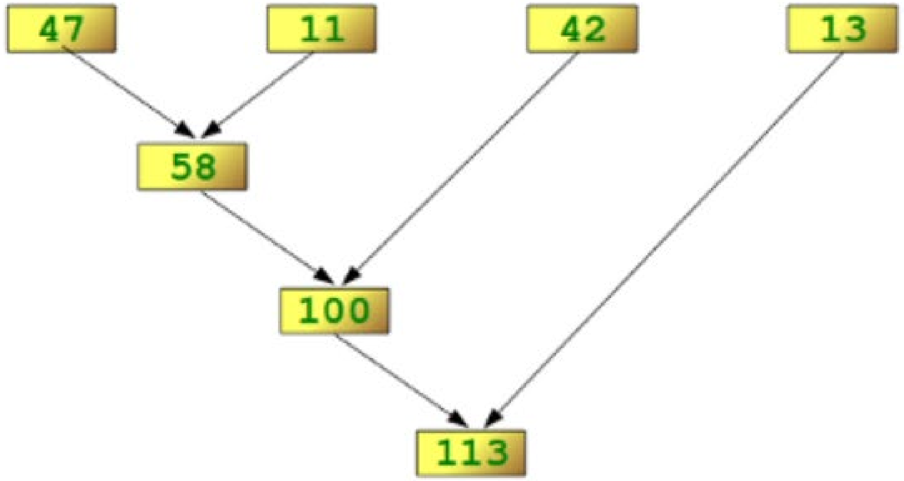
Reduce():

- The function reduce() continually applies the function to the sequence. It then returns a single value.
- Synatx: **reduce(function, sequence)**



```
In [1]:  from functools import reduce
         lst =[47,11,42,13]
         reduce(lambda x,y: x+y,lst)

Out[1]: 113
```

- For a better understanding of the process, look at the flow below.



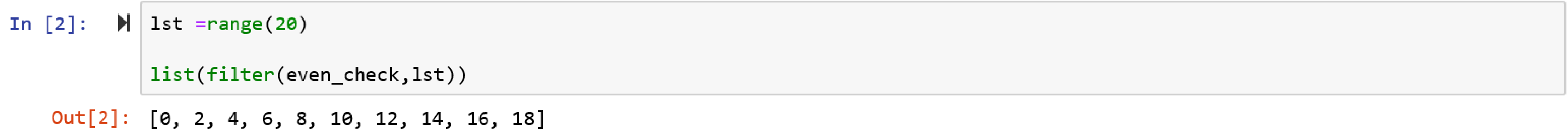
Python Built-in Functions

Filter(), zip()

Filter(): The function filter() offers a convenient way to filter out all the elements of an iterable, for which the function returns True.

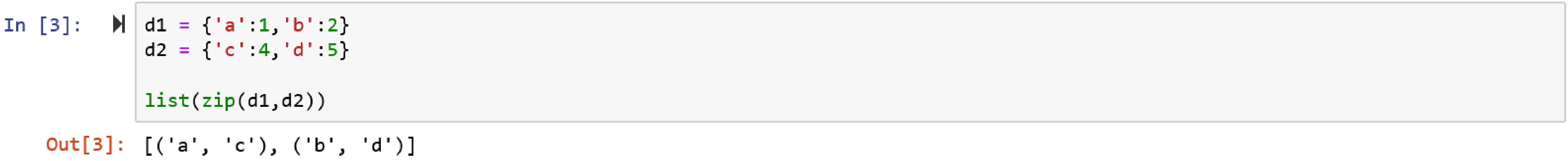
Syntax: **filter(function, list)**

- The function filter() needs a function as its first argument, which should return a Boolean value.



```
In [2]:  lst =range(20)

         list(filter(even_check,lst))

Out[2]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- **Zip():**



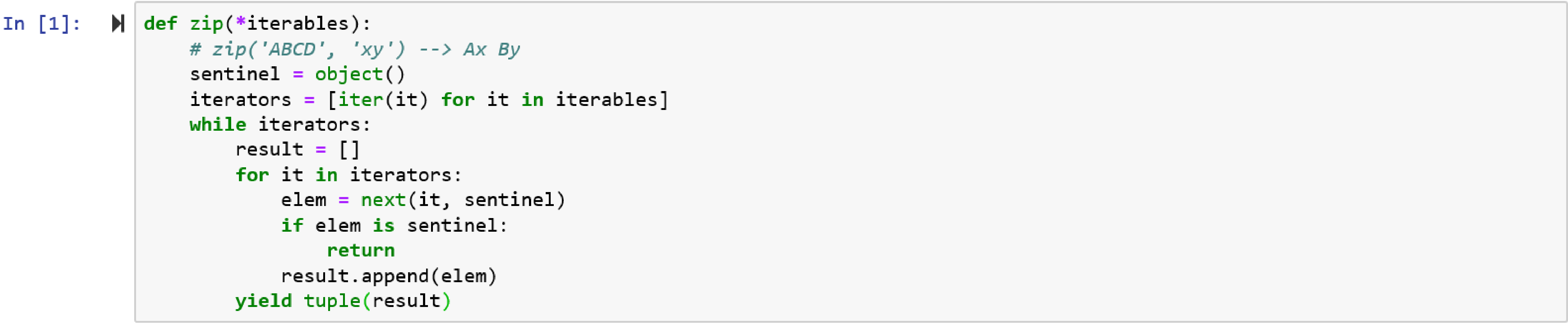
```
In [3]:  d1 = {'a':1,'b':2}
         d2 = {'c':4,'d':5}

         list(zip(d1,d2))

Out[3]: [('a', 'c'), ('b', 'd')]
```

Python Built-in Functions

Zip()

- **zip()** makes an iterator that aggregates elements from each of the iterables. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.

- zip() is equivalent to:



In [1]:

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

- zip() should only be used with unequal length inputs when you don’t care about trailing, unmatched values from the longer iterables.

Python Built-in Functions

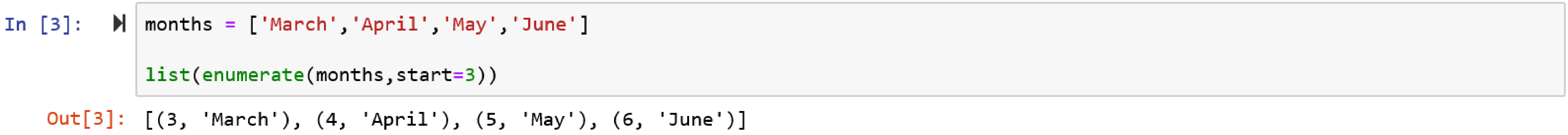
Enumerate()

Enumerate(): Enumerate allows us to keep a count as we iterate through an object. It does this by returning a tuple as (count,element).

- The function itself is equivalent to:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

- It takes an optional "start" argument to override the default value of zero



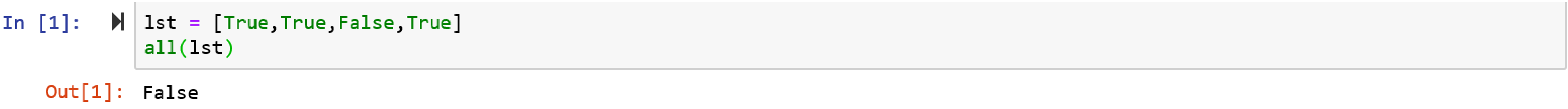
```
In [3]:  months = ['March','April','May','June']

         list(enumerate(months,start=3))

Out[3]: [(3, 'March'), (4, 'April'), (5, 'May'), (6, 'June')]
```

Python Built-in Functions

all(), any()

all() will return True if all elements in an iterable are True. It is the same as this function code:

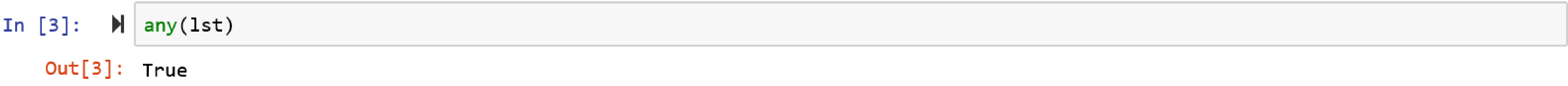
```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```



```
In [1]:  lst = [True,True,False,True]
         all(lst)

Out[1]:  False
```

any() will return True if any of the elements in the iterable are True. It is equivalent to the following function code:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```



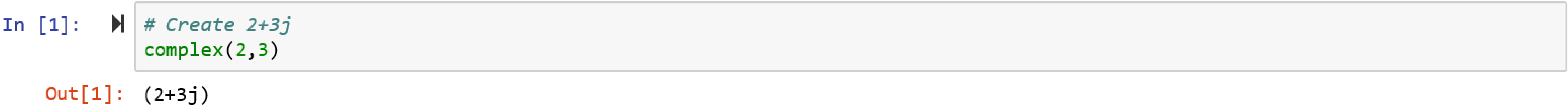
```
In [3]:  any(lst)

Out[3]:  True
```

Python Built-in Functions

complex()

COMPLEX():

- complex() returns a complex number with the value real + imag*1j or converts a string or number to a complex number.
- The second parameter can never be a string.
- Each argument may be any numeric type (including complex).
- If imag is omitted, it defaults to zero and the constructor serves as a numeric conversion like int and float.
- If both arguments are omitted, returns 0j.



```
In [1]:  # Create 2+3j
         complex(2,3)

Out[1]: (2+3j)
```

Contents

Manipulating DataFrames using pandas

- ➢ ***Indexing DataFrames***
- ➢ Slicing DataFrames
- ➢ Filtering DataFrames
- ➢ Transforming DataFrames

Indexing DataFrames

A Simple DataFrame

```
In [1]: import pandas as pd

In [2]: df = pd.read_csv('sales.csv',

index_col='month')  In [3]: df
Out[3]:
       eggs  salt  spam
month
Jan      47  12.0    17
Feb     110  50.0    31
Mar     221  89.0    72
Apr      77  87.0    20
May     132   NaN    52
Jun     205  60.0    55
```

Indexing DataFrames

Indexing using square brackets

```
In [4]: df
Out[4]:
       eggs  salt  spam
month
Jan      47  12.0    17
Feb     110  50.0    31
Mar     221  89.0    72
Apr      77  87.0    20
May     132   NaN    52
Jun     205  60.0    55

In [5]: df['salt']['Jan']
Out[5]: 12.0
```

Indexing DataFrames

Indexing using column attribute and row label

```
In [6]: df
Out[6]:
       eggs  salt  spam
month
Jan      47  12.0    17
Feb     110  50.0    31
Mar     221  89.0    72
Apr      77  87.0    20
May     132   NaN    52
Jun     205  60.0    55

In [7]: df.eggs['Mar']
Out[7]: 221
```

Indexing DataFrames

Indexing using .loc accessor

```
In [8]: df
Out[8]:
       eggs  salt  spam
month
Jan      47  12.0    17
Feb     110  50.0    31
Mar     221  89.0    72
Apr      77  87.0    20
May     132   NaN    52
Jun     205  60.0    55

In [9]: df.loc['May',
'spam']  Out[9]: 52.0
```

Indexing DataFrames

Indexing using .iloc accessor

```
In [10]: df
Out[10]:
       eggs  salt  spam
month
Jan      47  12.0    17
Feb     110  50.0    31
Mar     221  89.0    72
Apr      77  87.0    20
May     132   NaN    52
Jun     205  60.0    55

In [11]: df.iloc[4, 2]
Out[11]: 52.0
```

Indexing DataFrames

Selecting only some columns

```
In [12]: df_new =
df[['salt','eggs']]

In [13]: df_new
Out[13]:
       salt  eggs
month
Jan    12.0    47
Feb    50.0   110
Mar    89.0   221
Apr    87.0    77
May     NaN   132
Jun    60.0   205
```

Contents

Manipulating DataFrames using pandas

- ➢ Indexing DataFrames
- ➢ ***Slicing DataFrames***
- ➢ Filtering DataFrames
- ➢ Transforming DataFrames

Slicing DataFrames

Example of a sales data frame

```
In [1]: df
Out[1]:
       eggs  salt  spam
month
Jan      47  12.0    17
Feb     110  50.0    31
Mar     221  89.0    72
Apr      77  87.0    20
May     132   NaN    52
Jun     205  60.0    55
```

Slicing DataFrames

Selecting a column (i.e., series)

```
In [2]: df['eggs']
Out[2]:
month
Jan      47
Feb     110
Mar     221
Apr      77
May     132
Jun     205
Name: eggs, dtype: int64

In [3]: type(df['eggs'])
Out[3]: pandas.core.series.Series
```

Slicing DataFrames

Indexing and Slicing a Series

```
In [4]: df['eggs'][1:4] # Part of the eggs
column  Out[4]:
month
Feb    110
Mar    221
Apr     77
Name: eggs, dtype: int64

In [5]: df['eggs'][4]   # The value associated with May
Out[5]: 132
```

Slicing DataFrames

Using .loc[](1)

```
In [6]: df.loc[:, 'eggs':'salt'] # All rows, some
columns  Out[6]:
        eggs  salt
month
Jan       47  12.0
Feb      110  50.0
Mar      221  89.0
Apr       77  87.0
May      132   NaN
Jun      205  60.0
```

Slicing DataFrames

Using .loc[](2)

```
In [7]: df.loc['Jan':'Apr',:] # Some rows, all
columns  Out[7]:
       eggs  salt  spam
month
Jan      47  12.0    17
Feb     110  50.0    31
Mar     221  89.0    72
Apr      77  87.0    20
```

Slicing DataFrames

Using .loc[](3)

```
In [8]: df.loc['Mar':'May',
'salt':'spam']  Out[8]:
       salt  spam
month
Mar    89.0    72
Apr    87.0    20
May     NaN    52
```

Slicing DataFrames

Using .iloc()

```
In [9]: df.iloc[2:5, 1:] # A block from middle of the
DataFrame  Out[9]:
        salt  spam
month
Mar     89.0    72
Apr     87.0    20
May      NaN    52
```

Slicing DataFrames

Using lists rather than slices (1)

```
In [10]: df.loc['Jan':'May', ['eggs',
'spam']]  Out[10]:
       eggs  spam
month
Jan      47    17
Feb     110    31
Mar     221    72
Apr      77    20
May     132    52
```

Slicing DataFrames

Using Lists rather than Slices (2)

```
In [11]: df.iloc[[0,4,5],
0:2]  Out[11]:
        eggs  salt
month
Jan       47  12.0
May      132   NaN
Jun      205  60.0
```

Slicing DataFrames

Series Vs 1-Column Data frame

```
# A Series by column name
In [13]: df['eggs']
Out[13]:
month
Jan      47
Feb     110
Mar     221
Apr      77
May     132
Jun     205
Name: eggs, dtype: int64

In [14]:
type(df['eggs'])
Out[14]:
pandas.core.series.Series
```

```
# A DataFrame w/ single
column   In [15]: df[['eggs']]
Out[15]:
           eggs
month
Jan          47
Feb         110
Mar         221
Apr          77
May         132
Jun         205

In [16]:
type(df[['eggs']])
Out[16]:
pandas.core.frame.DataFrame
```

Contents

Manipulating DataFrames using pandas

- ➢ Indexing DataFrames
- ➢ Slicing DataFrames
- ➢ ***Filtering DataFrames***
- ➢ Transforming DataFrames

Filtering DataFrames

Creating a Boolean Series

```
In [1]: df.salt > 60
Out[1]:
month
Jan    False
Feb    False
Mar     True
Apr     True
May    False
Jun    False
Name: salt, dtype: bool
```

Filtering DataFrames

Filtering with a Boolean Series

```
In [2]: df[df.salt > 60]
Out[2]:
       eggs  salt  spam
month
Mar     221  89.0    72
Apr      77  87.0    20

In [3]: enough_salt_sold = df.salt > 60

In [4]:
df[enough_salt_sold]
Out[4]:eggs  salt  spam
month
Mar     221  89.0    72
Apr      77  87.0    20
```

Filtering DataFrames

Combining Filters

```
In [5]: df[(df.salt >= 50) & (df.eggs < 200)] # Both
conditions  Out[5]:
       eggs  salt  spam
month
Feb     110  50.0    31
Apr      77  87.0    20

In [6]: df[(df.salt >= 50) | (df.eggs < 200)] # Either
condition  Out[6]:
       eggs  salt  spam
month
Jan      47  12.0    17
Feb     110  50.0    31
Mar     221  89.0    72
Apr      77  87.0    20
May     132   NaN    52
Jun     205  60.0    55
```

Filtering DataFrames

Data frames with zeros and NaNs

```
In [7]: df2 = df.copy()

In [8]: df2['bacon'] = [0,
0, 50, 60, 70, 80]

In [9]: df2
Out[9]:


        eggs  salt  spam  bacon
month
Jan       47  12.0    17      0
Feb      110  50.0    31      0
Mar      221  89.0    72     50
Apr       77  87.0    20     60
May      132   NaN    52     70
Jun      205  60.0    55     80
```

Filtering DataFrames

Select columns with all nonzeros

```
In [10]: df2.loc[:,
df2.all()]  Out[10]:
       eggs  salt  spam
month
Jan      47  12.0    17
Feb     110  50.0    31
Mar     221  89.0    72
Apr      77  87.0    20
May     132   NaN    52
Jun     205  60.0    55
```

Filtering DataFrames

Select column with any nonzero

```
In [11]: df2.loc[:,
df2.any()]  Out[11]:
       eggs  salt  spam  bacon
month
Jan      47  12.0    17      0
Feb     110  50.0    31      0
Mar     221  89.0    72     50
Apr      77  87.0    20     60
May     132   NaN    52     70
Jun     205  60.0    55     80
```

Filtering DataFrames

Select columns with any NaNs

```
In [12]: df.loc[:,
df.isnull().any()]  Out[12]:
       salt
month
Jan    12.0
Feb    50.0
Mar    89.0
Apr    87.0
May     NaN
Jun    60.0
```

Filtering DataFrames

Select column without NaNs

```
In [13]: df.loc[:,
df.notnull().all()]  Out[13]:
       eggs  spam
month
Jan      47    17
Feb     110    31
Mar     221    72
Apr      77    20
May     132    52
Jun     205    55
```

Filtering DataFrames

Drop rows with NaNs

```
In [14]:
df.dropna(how='any')
Out[14]:eggs  salt  spam
month
Jan       47  12.0    17
Feb      110  50.0    31
Mar      221  89.0    72
Apr       77  87.0    20
Jun      205  60.0    55
```

Filtering DataFrames

Filtering a column based on another

```
In [15]: df.eggs[df.salt >
55]  Out[15]:
month
Mar    221
Apr     77
Jun    205
Name: eggs, dtype: int64
```

Filtering DataFrames

Modifying a column based on the other

```
In [16]: df.eggs[df.salt > 55] += 5

In [17]: df
Out[17]:
       eggs  salt  spam
month
Jan      47  12.0    17
Feb     110  50.0    31
Mar     226  89.0    72
Apr      82  87.0    20
May     132   NaN    52
Jun     210  60.0    55
```

Contents

Manipulating DataFrames using pandas

- Indexing DataFrames
- Slicing DataFrames
- Filtering DataFrames
- ***Transforming DataFrames***

Transforming DataFrames

Data frame vectorized methods

```
In [1]: df.floordiv(12)  # Convert to dozens unit
Out[1]:
       eggs  salt  spam
month
Jan       3   1.0     1
Feb       9   4.0     2
Mar      18   7.0     6
Apr       6   7.0     1
May      11   NaN     4
Jun      17   5.0     4
```

Transforming DataFrames

NumPy vectorized functions

```
In [2]: import numpy as np

In [3]: np.floor_divide(df,         # Convert to dozens unit
12)  Out[3]:
        eggs  salt  spam
month
Jan      3.0   1.0   1.0
Feb      9.0   4.0   2.0
Mar     18.0   7.0   6.0
Apr      6.0   7.0   1.0
May     11.0   NaN   4.0
Jun     17.0   5.0   4.0
```

Transforming DataFrames

Plain Python functions(1)

```
In [4]: def dozens(n):
   ....:     return n//12

In [5]: df.apply(dozens)  # Convert to dozens unit
Out[5]:
       eggs  salt  spam
month
Jan       3   1.0     1
Feb       9   4.0     2
Mar      18   7.0     6
Apr       6   7.0     1
May      11   NaN     4
Jun      17   5.0     4
```

Transforming DataFrames

Plain Python functions(2)

```
In [6]: df.apply(lambda n:
n//12)  Out[6]:
       eggs  salt  spam
month
Jan       3   1.0     1
Feb       9   4.0     2
Mar      18   7.0     6
Apr       6   7.0     1
May      11   NaN     4
Jun      17   5.0     4
```

Transforming DataFrames

Storing a transformation

```
In [7]: df['dozens_of_eggs'] =
df.eggs.floordiv(12)

In [8]: df
Out[8]:eggs  salt  spam  dozens_of_eggs
month
Jan      47  12.0    17               3
Feb     110  50.0    31               9
Mar     221  89.0    72              18
Apr      77  87.0    20               6
May     132   NaN    52              11
Jun     205  60.0    55              17
```

Transforming DataFrames

DataFrame index

```
In [9]: df
Out[9]:
       eggs  salt  spam  dozens_of_eggs
month
Jan      47  12.0    17               3
Feb     110  50.0    31               9
Mar     221  89.0    72              18
Apr      77  87.0    20               6
May     132   NaN    52              11
Jun     205  60.0    55              17

In [10]: df.index
Out[10]: Index(['Jan', 'Feb', 'Mar', 'Apr', 'May',
'Jun'],  dtype='object', name='month')
```

Transforming DataFrames

Working with string values(1)

```
In [11]: df.index = df.index.str.upper()

In [12]: df
Out[12]:
       eggs  salt  spam  dozens_of_eggs
month
JAN      47  12.0    17               3
FEB     110  50.0    31               9
MAR     221  89.0    72              18
APR      77  87.0    20               6
MAY     132   NaN    52              11
JUN     205  60.0    55              17
```

Transforming DataFrames

Working with string values(2)

```
In [13]: df.index =
df.index.map(str.lower)

In [14]: df
Out[14]:
     eggs  salt  spam  dozens_of_eggs
jan    47  12.0    17               3
feb   110  50.0    31               9
mar   221  89.0    72              18
apr    77  87.0    20               6
may   132   NaN    52              11
jun   205  60.0    55              17
```

Transforming DataFrames

Defining Columns using other columns

```
In [15]: df['salty_eggs'] = df.salt +
df.dozens_of_eggs

In [16]: df
Out[16]:
     eggs  salt  spam  dozens_of_eggs  salty_eggs
jan    47  12.0    17               3        15.0
feb   110  50.0    31               9        59.0
mar   221  89.0    72              18       107.0
apr    77  87.0    20               6        93.0
may   132   NaN    52              11         NaN
jun   205  60.0    55              17        77.0
```

Python & Oracle

Requirements

- **cx_Oracle** module

http://cx-oracle.sourceforge.net/

Installation

- Windows: Win Installer
- Linux: RPM or cx_Oracle.so

Example: accessing database

- To install cx_oracle

  python –m pip install cx_oracle
- To create a connection with database

  connection = cx_Oracle.connect(username/password@hostname:1521/XE’)

- Oracle database to local

```
import cx_Oracle
import pandas as pd

#to create a connection.
connection = cx_Oracle.connect('system/system@USHYDCAWASTHI4:1521/XE’)
cursor = connection.cursor()
cursor.execute("""select * from students""")

col1 = []
col2 = []
col3 = []
for STUDENT_NO, SURNAME,FORENAME in cursor:
    col1.append(STUDENT_NO)
    col2.append(SURNAME)
    col3.append(FORENAME)
    print("Values:", STUDENT_NO, SURNAME,FORENAME)
```

Example: accessing database

```
df = pd.DataFrame()
df['STUDENT_NO'] = col1
df['SURNAME'] = col2
df['FORENAME'] = col3

df.to_csv(“path”+database_file.csv)
```

- Python to Oracle database

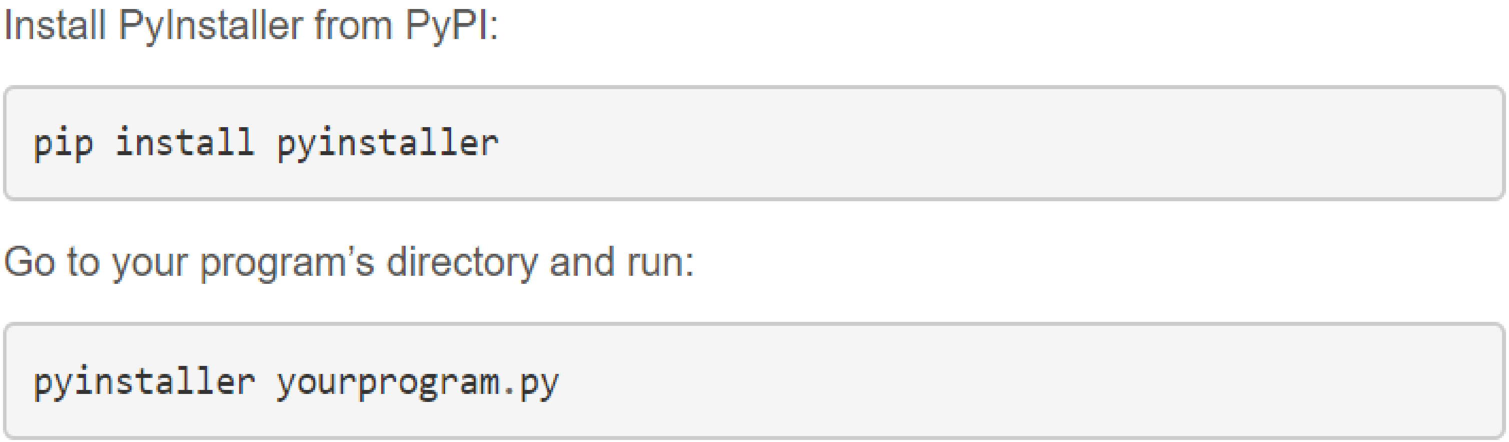
```
rows = [ (1, "First" ),
         (2, "Second" ),
         (3, "Third" ),
         (4, "Fourth" ),
         (5, "Fifth" ),
         (6, "Sixth" ),
         (7, "Seventh" ) ]

cur = connection.cursor()
cur.bindarraysize = 7
cur.setinputsizes(int, 20)
cur.executemany("insert into sample(id, data) values (:1, :2)", rows)
connection.commit()
```

Python & Pyinstaller

PyInstaller freezes (packages) Python applications into stand-alone executables, under Windows, GNU/Linux, Mac OS X, FreeBSD, Solaris and AIX

PyInstaller’s main advantages over similar tools are that PyInstaller works with Python 2.7 and 3.5—3.7, it builds smaller executables thanks to transparent compression, it is fully multi-platform, and use the OS support to load the dynamic libraries, thus ensuring full compatibility.

PyInstaller Quickstart



Install PyInstaller from PyPI:

```
pip install pyinstaller
```

Go to your program’s directory and run:

```
pyinstaller yourprogram.py
```

Steps to Create an Executable from Python Script using Pyinstaller

- Step 1: Open the Windows Command Prompt
- Step 2: Install the Pyinstaller Package

```
pip install pyinstaller
```

- Step 3: Save your Python Script
- Step 4: Create the Executable using Pyinstaller

```
pyinstaller --onefile pythonscript.py
```