week1_lecture

June 22, 2020

1 Week 1 - Recap of basic Python and Jupyter concepts

1.0.1 Aims

- To introduce Jupyter Notebook that we will be using, to write Python scripts, in this course
- To recapitulate Python basic functionality

1.0.2 Learning objectives

- Learn about the functions in the various toolbars in Jupyter
- Become familiar with creating and modifying cells in Jupyter to write code and text
- Review data types and data structures in Python
- Present ways of writing if/while conditions and comparisons in Python
- Show ways to read and write files in Python

2 Running Python within a Jupyter Notebook

2.0.1 Starting Jupyter Notebook

For details about installing Python3 and Jupyter, please refer to the Week 0 - Getting ready - Introduction and installations. If you happen to have issues with the installation, Sam and Dan will be available to help after today's session.

Once you have Python3 and Jupyter installed, start Jupyter Notebook from the icon in Windows start menu:

You should now see Jupyter Notebook home page loading in your default web browser e.g. Chrome, Firefox …

2.0.2 Downloading today's notebook

All the materials that we will use throughout the course will be made available in this GitHub repository:

https://github.com/semacu/data-science-python

To download today's notebook, go to your web browser and paste the url above:

1. Click on the `notebooks` folder and then click on the notebook `week1_lecture.ipynb`. It may take a few seconds to load, then click on `Raw`.
2. Press `Ctrl+S` or right-click on `Download as ...` or `Save as ...` to save it as `week1_lecture.ipynb`

3. Go to the Jupyter Notebook that you started earlier and navigate to the location where you saved `week1_lecture.ipynb`
4. Open the file. You will now see the code.

2.0.3 Navigating notebooks

The dashboard of notebooks This is the default page when you open Jupyter notebook a.k.a. the dashboard. Its main purpose is to display the directories and notebooks present in the current directory, e.g.:

By clicking on the directories or files in the list, you can navigate your file system.

To create a new notebook, click on the "New" button at the top right of the list and select `Python 3` from the dropdown menu (as seen below). The versions of Python listed depends on what's installed on your machine.

Notebooks and files created locally can be uploaded to the current directory by dragging them onto the list.

The notebooks are highlighted in green if they are currently "Running". Notebooks remain running until you explicitly shut them down; closing the notebook's page is not sufficient.

To shutdown, delete, duplicate, or rename a notebook check the checkbox next to it and an array of controls will appear at the top of the notebook list.

To see all of your running notebooks along with their directories, click on the "Running" tab:

This view provides a convenient way to track notebooks that you start as you navigate the file system.

The notebook user interface If you create a new notebook or open an existing one, you will be taken to the notebook user interface. This user interface allows you to run code and author notebook documents interactively. The user interface has the following main areas:

- Menubar
- Toolbar
- Notebook area and cells

The notebook has an interactive tour of these elements that can be started in the "Help:User Interface Tour" menu item - please follow the tour:

Modes The keyboard does different things depending on which mode the notebook is in. There are two modes:

- Edit mode: indicated by a green cell border and a prompt showing in the editor area. When a cell is in edit mode, you can type into the cell, like a normal text editor. Enter the edit mode by clicking on a cell's area or press `Enter`.
- Command mode: indicated by a grey cell border with a blue left margin. You are able to edit the cells as a whole, but not type into individual cells. Enter the command mode by clicking *outside* a cell's editor area or press `Esc`.

3 Basic Python

3.0.1 Printing

You can include a comment in python by prefixing some text with a `#` character. All text following the `#` will then be ignored by the interpreter.

[ ]:
```
print('Hello from python!') # to print some text, enclose it between single␣
 ↪quotation marks
print("I'm here today!")    # or double quotation marks
print(34)                   # print an integer
print(2 + 4)                # print the result of an arithmetic operation
print("The answer is", 42)  # print multiple expressions, separated by comma
```

3.0.2 Variables

A variable can be assigned to a simple value or the outcome of a more complex expression. The `=` operator is used to assign a value to a variable.

[ ]:
```
x = 3     # assignment of a simple value
print(x)
y = x + 5 # assignment of a more complex expression
print(y)
i = 12
print(i)
i = i + 1 # assigment of the current value of a variable incremented by 1 to␣
 ↪itself
print(i)
i += 1    # shorter version with the special += operator
print(i)
```

3.0.3 Simple data types

Python has four main data types: integer, float, string and boolean

[ ]:
```
a = 2           # integer
b = 5.0         # float
c = 'word'      # string
d = 4 > 5       # boolean (True or False)
e = None        # special built-in value to create a variable that has not been␣
 ↪set to anything specific
print(a, b, c, d, e)
print(a, 'is of type', type(a)) # to check the type of a variable
```

3.0.4 Arithmetic operations

Using Python as a calculator to do simple mathematical operations in integer and float numbers

[ ]:
```
a = 2              # assignment
a += 1             # change and assign (*=, /=)
3 + 2              # addition
3 - 2              # subtraction
3 * 2              # multiplication
3 / 2              # integer (python2) or float (python3) division

3 // 2             # integer division
3 % 2              # remainder
3 ** 2             # exponent
```

3.0.5 Data structures

There are also different types of data structures in Python to store the main basic data types shown above:

- Lists
- Sets
- Tuples
- Dictionaries

Lists A list is an ordered collection of mutable elements e.g. the elements inside a list can be modified.

[ ]:
```
a = ['red', 'blue', 'green']        # manual initialisation
copy_of_a = a[:]                    # copy of a
another_a = a                       # same as a
b = list(range(5))                  # initialise from iteratable
c = [1, 2, 3, 4, 5, 6]              # manual initialisation
len(c)                              # length of the list
d = c[0]                            # access first element at index 0
e = c[1:3]                          # access a slice of the list,
                                    # including element at index 1 up to but not␣
 ↪including element at index 3
f = c[-1]                           # access last element
c[1] = 8                            # assign new value at index position 1
g = ['re', 'bl'] + ['gr']           # list concatenation
['re', 'bl'].index('re')            # returns index of 're'
a.append('yellow')                  # add new element to end of list
a.extend(b)                         # add elements from list `b` to end of list␣
 ↪`a`
a.insert(1, 'yellow')               # insert element in specified position
're' in ['re', 'bl']                # true if 're' in list
'fi' not in ['re', 'bl']            # true if 'fi' not in list
c.sort()                            # sort list in place
h = sorted([3, 2, 1])               # returns sorted list
i = a.pop(2)                        # remove and return item at index (default␣
 ↪last)
```

```
print(a)
print(b)
print(c)
print(d)
print(e)
print(f)
print(g)
print(h)
print(i)
print("------------")
print(a)
print(copy_of_a)
print(another_a)
```

Sets A set is an unordered collection of unique elements.

[ ]:
```
a = {1, 2, 3}                                   # initialise manually
b = set(range(5))                               # initialise from iteratable
c = set([1,2,2,2,2,4,5,6,6,6])                  # initialise from list
a.add(13)                                       # add new element to set
a.remove(13)                                    # remove element from set
2 in {1, 2, 3}                                  # true if 2 in set
5 not in {1, 2, 3}                              # true if 5 not in set
d = a.union(b)                                  # return the union of sets as a␣
 ↪new set
e = a.intersection(b)                           # return the intersection of sets␣
 ↪as a new set
print(a)
print(b)
print(c)
print(d)
print(e)
```

Tuples A tuple is an ordered collection of immutable elements e.g. tuples are similar to lists, but the elements inside a tuple cannot be modified. Most of the list operations shown above can be used on tuples as well, with the exception of the assignment of new value at a certain index position.

[ ]:
```
a = (123, 54, 92)               # initialise manually
b = ()                          # empty tuple
c = ("Ala",)                    # tuple of a single string (note the trailing␣
 ↪",")
d = (2, 3, False, "Arg", None) # a tuple of mixed types
print(a)
print(b)
print(c)
```

```
print(d)
t = a, c, d                    # tuple packing
x, y, z = t                    # tuple unpacking
print(t)
print(x)
print(y)
print(z)
```

Dictionaries A dictionary is an unordered collection of key-value pairs where keys must be unique.

[ ]:
```
a = {'A': 'Adenine', 'C': 'Cytosine'}        # dictionary
b = a['A']                                   # translate item
c = a.get('N', 'no value found')             # return default value
'A' in a                                     # true if dictionary a contains␣
 ↪key 'A'
a['G'] = 'Guanine'                           # assign new key, value pair to␣
 ↪dictonary a
a['T'] = 'Thymine'                           # assign new key, value pair to␣
 ↪dictonary a
print(a)
print(b)
print(c)
d = a.keys()                                 # get list of keys
e = a.values()                               # get list of values
f = a.items()                                # get list of key-value pairs
print(d)
print(e)
print(f)
del a['A']                                   # delete key and associated value
print(a)
```

3.0.6 Working with strings

A string is an ordered collection of immutable characters or tuple of characters.

[ ]:
```
a = 'red'                           # assignment
char = a[2]                         # access individual characters
b = 'red' + 'blue'                  # string concatenation
c = '1, 2, three'.split(',')        # split string into list
d = '.'.join(['1', '2', 'three'])   # concatenate list into string
print(a)
print(char)
print(b)
print(c)
print(d)
dna = 'ATGTCACCGTTT'                # assignment
```

```
seq = list(dna)                          # convert string into list of character
e = len(dna)                             # return string length
f = dna[2:5]                             # slice string
g = dna.find('TGA')                      # substring location, return -1 when not␣
 ↪found
print(dna)
print(seq)
print(e)
print(f)
print(g)
text = '   chrom start end    '          # assignment
print('>', text, '<')
print('>', text.strip(), '<')            # remove unwanted whitespace at both end of␣
 ↪the string
print('{:.2f}'.format(0.4567))           # formating string
print('{gene:s}\t{exp:+.2f}'.format(gene='Beta-Actin', exp=1.7))
```

3.0.7 Conditions

A conditional `if` or `elif` statement is used to specify that some block of code should only be executed if a conditional expression is **True**. Often the final `else` statement works when all the conditions before are **False**.

Python uses indentation to represent which statements are inside a block of code e.g. the line after the `if` statement is indented (tab).

[ ]:
```
a, b = 1, 2              # assign different values to a and b, and execute the␣
 ↪cell to test
if a + b == 3:
    print('Three')
elif a + b == 1:
    print('One')
else:
    print('?')
```

3.0.8 Comparisons

[ ]:
```
1 == 1            # equal
1 != 2            # not equal
2 > 1             # greater than
2 < 1             # smaller than

1 != 2 and 2 < 3  # logical AND
1 != 2 or 2 < 3   # logical OR
not 1 == 2        # logical NOT

a = list('ATGTCACCGTTT')
```

```
b = a               # same as a
c = a[:]            # copy of a
'N' in a            # test if character 'N' is in a

print('a', a)       # print a
print('b', b)       # print b
print('c', c)       # print c
print('Is N in a?', 'N' in a)
print('Are objects b and a point to the same memory address?', b is a)
print('Are objects c and a point to the same memory address?', c is a)
print('Are values of b and a identical?', b == a)
print('Are values of c and a identical?', c == a)
a[0] = 'N'          # modify a
print('a', a)       # print a
print('b', b)       # print b
print('c', c)       # print c
print('Is N in a?', 'N' in a)
print('Are objects b and a point to the same memory address?', b is a)
print('Are objects c and a point to the same memory address?', c is a)
print('Are values of b and a identical?', b == a)
print('Are values of c and a identical?', c == a)
```

3.0.9 Loops

There are two ways of creating loops in Python using `for` or `while`.

for

[ ]:
```
a = ['red', 'blue', 'green']
for color in a:
    print(color)
```

while

[ ]:
```
number = 1
while number < 10:
    print(number)
    number += 1
```

Python has two ways of affecting the flow of a `for` or `while` loop:

- The `break` statement immediately causes all looping to finish, and execution is resumed at the next statement after the loop.
- The `continue` statement means that the rest of the code in the block is skipped for this particular item in the collection.

[ ]:
```
# break
sequence = ['CAG','TAC','CAA','TAG','TAC','CAG','CAA']
for codon in sequence:
```

```
    if codon == 'TAG':
        break            # Quit the looping at this point (the TAG stop codon)
    else:
        print(codon)


# continue
values = [10, -5, 3, -1, 7]
total = 0
for v in values:
    if v < 0:
        continue         # Don't quit the loop but skip the iterations where␣
 ↪the integer is negative
    total += v

print(values, 'sum:', sum(values), 'total:', total)
```

3.0.10 Reading and writing files

To read from a file, your program needs to `open` the file and then read the contents of the file. You can read the entire contents of the file at once, or read the file line by line. The `with` statement makes sure the file is closed properly when the program has finished accessing the file.

We will use one of the files containing clinical and genomic data from anonymized patients published as part of the METABRIC consortium, a large-scale collaborative study of breast cancer worldwide (PMID: 22522925).

Note: you can download the data file `metabric_clinical_and_expression_data.csv` from the `data` folder in: https://github.com/semacu/data-science-python

[ ]:
```
# reading from file
with open("../data/metabric_clinical_and_expression_data.csv") as f:
    for line in f:
        print(line.strip())
```

Passing the `w` argument in `open()` tells Python that you want to create and write to a new file.

[ ]:
```
# writing to a file
with open('programming.txt', 'w') as f:
    f.write("I love programming in Python!\n")
    f.write("I love making scripts.\n")
```

Keep in mind this will erase the contents of the file if it already exists.

Passing the `a` argument instead tells Python you want to append to the end of an existing file.

[ ]:
```
# appending to a file
with open('programming.txt', 'a') as f:
    f.write("I love working with data.\n")
```

3.0.11 Getting help

The Python Standard Library is the reference documentation of all libraries included in Python as well as built-in functions and data types.

[ ]:
```
help(len)           # help on built-in function
```

[ ]:
```
help(list.extend)  # help on list function
```

For help within the Jupyter Notebook, try the following:

[ ]:
```
len?
```

3.0.12 Assignment

1. Create a new Jupyter Notebook with one Markdown cell and one Code cell and name it as you wish e.g. `week1_exercise1`
   - Print out the names of the amino acids that would be produced by the DNA sequence "GTT GCA CCA CAA CCG" - see the DNA codon table here. Note: split the string into the individual codons and then create and use a dictionary to map between codon sequences and the amino acids they encode
   - Print each codon and its corresponding amino acid
   - Why couldn't we build a dictionary where the keys are names of amino acids and the values are the DNA codons?
   - Download the python file associated with the notebook you have created
2. You are going to look at the METABRIC data file `metabric_clinical_and_expression_data.csv` on breast cancer referred above
   - Write a script that reads the file and counts how many unique patients we have data available
   - How many patients were older than 75 when diagnosed with breast cancer?
   - What were the earliest and oldest age of diagnosis?
   - Count how many patients were treated with Chemotherapy and Radiotherapy
   - Count how many patients had less than three mutations in the genes investigated
3. `Bonus exercise`
   - Starting with an empty dictionary, count the abundance of different residue types present in the 1-letter lysozyme protein sequence and print the results to the screen in alphabetical key order.
   - Write the results to an output file