01a - Introduction - Python

January 15, 2017

1 Python for data analysis

For those who are new to using Python for scientific work, we first provide a short introduction to Python and the most useful packages for data analysis.

```
In [1]: from preamble import * # Ignore, this is just to make code cleaner
        HTML('''<style>.CodeMirror{min-width:100% !important;}</style>''') # For slides
```

```
Out[1]: <IPython.core.display.HTML object>
```

1.1 Python

Disclaimer: We can only cover some of the basics here. If you are completely new to Python, we recommend to take an introductory online course, such as the DataCamp Intro to Python for Data Science.

1.1.1 Hello world

- Printing is doen with the print() function.
- Everything after # is considered a comment.
- You don't need to end commands with ';'.

```
In [2]: # This is a comment
        print("Hello world")
        print(5 / 8)
        5/8 # This only prints in IPython notebooks and shells.
```

```
Hello world
0.625
```

```
Out[2]: 0.625
```

Note: In these notebooks we'll use Python interactively to avoid having to type print() every time.

1.1.2 Basic data types

Python has all the basic data types and operations: int, float, str, bool, None.
Variables are **dynamically typed**: you need to give them a value upon creation, and they will have the data type of that value. If you redeclare the same variable, if will have the data type of the new value.
You can use type() to get a variable's type.

```
In [3]: s = 5
        type(s)
        s > 3 # Booleans: True or False
        s = "The answer is "
        type(s)
```

```
Out[3]: int
```

```
Out[3]: True
```

```
Out[3]: str
```

Python is also **strongly typed**: it won't implicitly change a data type, but throw a TypeError instead. You will have to convert data types explictly, e.g. using str() or int().
Exception: Arithmetic operations will convert to the *most general* type.

```
In [4]: 1.0 + 2     # float + int -> float
        s + str(42) # string + string
        # s + 42    # Bad: string + int
```

```
Out[4]: 3.0
```

```
Out[4]: 'The answer is 42'
```

1.1.3 Complex types

The main complex data types are lists, tuples, sets, and dictionaries (dicts).

```
In [5]: l = [1,2,3,4,5,6]       # list
        t = (1,2,3,4,5,6)       # tuple: like a list, but immutable
        s = set((1,2,3,4,5,6))  # set: unordered, you need to use add() to add new elements
        d = {2: "a",            # dict: has key - value pairs
             3: "b",
             "foo": "c",
             "bar": "d"}

        l  # Note how each of these is printed
        t
        s
        d
```

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

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

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

```
Out[5]: {'foo': 'c', 2: 'a', 3: 'b', 'bar': 'd'}
```

You can use indices to return a value (except for sets, they are unordered)

```
In [6]: l
        l[2]
        t
        t[2]
        d
        d[2]
        d["foo"]
```

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

```
Out[6]: 3
```

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

```
Out[6]: 3
```

```
Out[6]: {'foo': 'c', 2: 'a', 3: 'b', 'bar': 'd'}
```

```
Out[6]: 'a'
```

```
Out[6]: 'c'
```

You can assign new values to elements, except for tuples

```
In [7]: l
        l[2] = 7 # Lists are mutable
        l
        # t[2] = 7 # Tuples are not
```

```
Out[7]: [1, 2, 3, 4, 5, 6]
```

```
Out[7]: [1, 2, 7, 4, 5, 6]
```

Python allows convenient tuple packing / unpacking

```
In [8]: b = ("Bob", 19, "CS")       # tuple packing
        (name, age, studies) = b    # tuple unpacking
        name
        age
        studies
```

```
Out[8]: 'Bob'
```

```
Out[8]: 19
```

```
Out[8]: 'CS'
```

1.1.4 Strings

Strings are quite powerful.
They can be used as lists, e.g. retrieve a character by index.
They can be formatted with the format operator (%), e.g. %s for strings, %d for decimal integers, %f for floats.

```
In [9]: s = "The %s is %d" % ('answer', 42)
        s
        s[0]
        s[4:10]
        '%.2f' % (3.14159265) # defines number of decimal places in a float

Out[9]: 'The answer is 42'

Out[9]: 'T'

Out[9]: 'answer'

Out[9]: '3.14'
```

They also have a format() function for more complex formatting

```
In [10]: l = [1,2,3,4,5,6]
         "{}".format(l)
         "%s" % l       # This is identical
         "{first} {last}".format(**{'first': 'Hodor', 'last': 'Hodor!'})

Out[10]: '[1, 2, 3, 4, 5, 6]'

Out[10]: '[1, 2, 3, 4, 5, 6]'

Out[10]: 'Hodor Hodor!'
```

1.1.5 For loops, If statements

For-loops and if-then-else statements are written like this.
Indentation defines the scope, not brackets.

```
In [11]: l = [1,2,3]
         d = {"foo": "c", "bar": "d"}

         for i in l:
             print(i)

         for k, v in d.items(): # Note how key-value pairs are extracted
             print("%s : %s" % (k,v))

         if len(l) > 3:
             print('Long list')
         else:
             print('Short list')
```

```
1
2
3
foo : c
bar : d
Short list
```

1.1.6 Functions

Functions are defined and called like this:

```
In [12]: def myfunc(a, b):
             return a + b

         myfunc(2, 3)
```

```
Out[12]: 5
```

Function arguments (parameters) can be: * variable-length (indicated with *) * a dictionary of keyword arguments (indicated with **). * given a default value, in which case they are not required (but have to come last)

```
In [13]: def func(*argv, **kwarg):
             print("func argv: %s" % str(argv))
             print("func kwarg: %s" % str(kwarg))

         func(2, 3, a=4, b=5)

         def func(a=2):
             print(a * a)

         func(3)
         func()
```

```
func argv: (2, 3)
func kwarg: {'a': 4, 'b': 5}
9
4
```

Functions can have any number of outputs.

```
In [14]: def func(*argv):
             return sum(argv[0:2]), sum(argv[2:4])

         sum1, sum2 = func(2, 3, 4, 5)
         sum1, sum2
```

```
def squares(limit):
    r = 0
    ret = []

    while r < limit:
        ret.append(r**2)
        r += 1

    return ret

for i in squares(4):
    print(i)
```

```
Out[14]: (5, 9)
```

```
0
1
4
9
```

Functions can be passed as arguments to other functions

```
In [15]: def greet(name):
             return "Hello " + name

         def call_func(func):
             other_name = "John"
             return func(other_name)

         call_func(greet)
```

```
Out[15]: 'Hello John'
```

Functions can return other functions

```
In [16]: def compose_greet_func():
             def get_message():
                 return "Hello there!"

             return get_message

         greet = compose_greet_func()
         greet()
```

```
Out[16]: 'Hello there!'
```

1.1.7 Classes

Classes are defined like this

```
In [17]: class TestClass(object): # TestClass inherits from object.
             myvar = ""

             def __init__(self, myString): # optional constructor, returns nothing
                 self.myvar = myString # 'self' is used to store instance properties

             def say(self, what): # you need to add self as the first argument
                 return self.myvar + str(what)

         a = TestClass("The new answer is ")
         a.myvar # You can retrieve all properties of self
         a.say(42)

Out[17]: 'The new answer is '

Out[17]: 'The new answer is 42'
```

Static functions need the @staticmethod decorator

```
In [18]: class TestClass(object):
             myvar = ""

             def __init__(self, myString):
                 self.myvar = myString

             def say(self, what): # you need to add self as the first argument
                 return self.myvar + str(what)

             @staticmethod
             def sayStatic(what): # or declare the function static
                 return "The answer is " + str(what)

         a = TestClass("The new answer is ")
         a.say(42)
         a.sayStatic(42)

Out[18]: 'The new answer is 42'

Out[18]: 'The answer is 42'
```

1.1.8 Functional Python

You can write complex procedures in a few elegant lines of code using built-in functions and libraries such as functools, itertools, operator.

```
In [19]: def square(num):
             return num ** 2

         # map(function, iterable) applies a given function to every element of a list
         list(map(square, [1,2,3,4]))

         # a lambda function a is function created on the fly
         list(map(lambda x: x**2, [1,2,3,4]))
         list(map(lambda x: x if x>2 else 0, [1,2,3,4]))

         # reduce(function, iterable) applies a function with two arguments cumulatively to ever
         from functools import reduce
         reduce(lambda x,y: x+y, [1,2,3,4])
```

```
Out[19]: [1, 4, 9, 16]

Out[19]: [1, 4, 9, 16]

Out[19]: [0, 0, 3, 4]

Out[19]: 10
```

```
In [20]: # filter(function, iterable)) extracts every element for which the function returns tru
         list(filter(lambda x: x>2, [1,2,3,4]))

         # zip([iterable,...]) returns tuples of corresponding elements of multiple lists
         list(zip([1,2,3,4],[5,6,7,8,9]))
```

```
Out[20]: [3, 4]

Out[20]: [(1, 5), (2, 6), (3, 7), (4, 8)]
```

list comprehensions can create lists as follows:

```
[statement for var in iterable if condition]
```

generators do the same, but are lazy: they don't create the list until it is needed:

```
(statement for var in list if condition)
```

```
In [21]: a = [2, 3, 4, 5]

         lc = [ x for x in a if x >= 4 ] # List comprehension. Square brackets
         lg = ( x for x in a if x >= 4 ) # Generator. Round brackets

         a.extend([6,7,8,9])

         for i in lc:
             print("%i " % i, end="") # end tells the print function not to end with a newline
         print("\n")
         for i in lg:
             print("%i " % i, end="")
```

```
4 5

4 5 6 7 8 9
```