Course Introduction and Python Basics

Johan Falkenjack

2018-01-15

Course Introduction

- Python > Data Science
- 3 hp
- pass/fail
- 3 lectures, 3 labs (2 sessions for each lab)
  - Python Basics
  - Programming Paradigms in Python
  - Data Science in Python
- Two teachers
  - Eva Ragnemalm (examiner and lab assistant)
  - Johan Falkenjack (course leader and lecturer)

First lecture

- What is Python?
- Python development environments
- Data types in Python
- Program flow and control statements
  - Conditionals
  - Loops
  - Subroutines (functions)
  - Modules (libraries and packages)
- Debugging

What is Python?

- First version in 1991
- **High-level language**
- Emphasizes readability
- **Interpreted/JIT-compiled** [compilation with Cython optional]
- Automatic memory management
- Strongly dynamically typed
- **Functional and/or object-oriented**
- **Glue** to other programs (interface to C/C++ or Java etc)
- Data Science ("Prototype in R, implement in Python")
- Two currently developed versions, 2.x and 3.x
  - **This course uses Python 3**

The Benevolent(?) Dictator For Life (BDFL) Guido van Rossum

Python development environments

- Python REPL and IPython
- Any text editor and command line
- IDLE
- General IDEs (PyCharm, Eclipse, Visual Studio, etc.)
- Scientific IDEs (Spyder, Rodeo, etc.)
- Notebooks (Jupyter)
  - **This course uses Jupyter Notebooks**

Python distributions

- Full distributions
  - CPython
  - Python(x,y)
  - Anaconda
  - PyPy
  - Jython
- Package managers
  - pip
  - conda

Python peculiarites (compared to R/Matlab)

- Not primarily a numerical language, e.g. no built-in vectors, matrices and operations on similar data types might not do what you expect.
- **Indexing begins at 0**, as indexes refer to breakpoints between elements.
- **Indentation matters!**
- Can import specific functions from a module.
- Assignment **by object**, **NOT by copy** or **by reference**.
  - Approximately, assignment **by copy of reference**.
- `a = b = 1` assigns 1 to both `a` and `b`.
- No exact equivalent to R's catch-all list type, which might actually be a good thing.

Python's objects

- Built-in types: **numbers**, **strings**, **lists**, **dictionaries**, **tuples**, **files**, etc.
- **Everything** is an object and has a class, i.e. no primitive data types.
- Python is a **strongly typed** language. `'johan' + 3` gives an error.
- Python is a **dynamically typed** language. No need to declare variables' type before they are used. Python figures out the object's type.
- Implication: Polymorphism by default:
  - "In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with." - Alex Martelli

Strings

- No character type, strings are strings, even if they are length 1.
- `s = 'Spam'`
- `len(s)` returns the number of letters.
- `s.lower(), s.upper(), s.count('m'), s.endswith('am'), ...`
- **Which methods are available for my object?** Try in Jupyter: `type s.` followed by TAB.
- $+$ operator **concatenates strings** by creating a new string (computationally expensive if you do it often).
- `sentence = 'Guido is the benevolent dictator for life'.sentence.split()`
- `s*3` returns `'SpamSpamSpam'`

Indexing strings

- Strings are always read from left to right
- Indexes are the positions between characters and start at 0, i.e. the position before the first character.
- `s[0]` starts at index 0 and reads once character forward, i.e. returns the first character in s.
- Indexes can be negative and are then counted from the end of the sequence (they are **NOT** excluders as in R).
- `s[-2]` returns next to last character.

Slicing strings

- Slicing is when you take a "slice" from a string based on indexes, for instance `s[1:3]` returns the substring from index 1 to 3, i.e. the second and third characters.
- s[:3] returns the string up to index 3, s[3:] returns the string from index 3 to the end.
- s[2] is equivalent to s[2:3] (with strings).
- `s[-3:]` returns the substring containing the last three characters.

String formatting

- String formatting is often used to create recurrent string injections:
- "My name is {0}".format('Fred')
- "The story of {0}, {1}, and {c}".format(a, b, c=d)

The list object

- A list is an ordered **container of several values**, possibly of different types.
- `myList = ['spam','spam','bacon',2]`
- The list object has several associated **methods**
  - `myList.append('egg')`
  - `myList.count('spam')`
  - `myList.sort()`
- $+$ operator concatenates lists: `myList + myOtherList` merges the two lists as one list.
- Elements are **not named** and lists are **not vectors** in a mathematical sense.

The list object

- Extract elements from a list: `myList[1]`
- Lists inside lists (nested lists):
  - `myOtherList = ['monty','Python']`
  - `myList[1] = myOtherList`
  - `myList[1]` returns the list `['monty','Python']`
  - `myList[1][1]` returns the string 'Python'
- Indexing and slicing works as with strings...
  - **EXCEPT** a bit more generally, in a string, all elements are strings, but not all elements of lists are necessarily lists.
  - myOtherList[1] returns an element (the string 'Python') while myOtherList[1:2] returns a list (['Python']).

Using lists to be more efficient with strings

- Strings are **immutable**, i.e. can't be changed after creation.
- Every "change" has to create a new string (remember concatenation).
- Try to avoid creating more strings than necessary:
  - Avoid: `my_string = 'Python' + ' ' + 'is' + ' ' + 'fun!'`
  - Instead: `my_string = ' '.join(['Python', 'is', 'fun!'])`
- In loops where you construct strings, add constituents to a list and join after loop finishes.
- Remember that you have to explicitly change the type of, for instance, numbers, when concatenating: str(1)

Dictionaries

- **Unordered** collection of pairs, most often names and some value, but are also often used to represent sparse data.
- `myDict = {'Sarah':29, 'Erik':28, 'Evelina':26}`
- Elements are **accessed by key not by index**: `myDict['Evelina']` returns 26.
- **Values can contain any object**: `myDict = {'Marcus':[23,14], 'Cassandra':17, 'Per':[12,29]}`. `myDict['Marcus'][1]` returns 14.
- **Keys must be immutable**: `myDict = {2:'contents of box2', (3, 'a'):'content of box 4', 'blackbox':10}`
- `myDict.keys()`, `myDict.values()`, `myDict.items()`

Tuples

- `myTuple = (3,4,'johan')`
- **Like lists, but immutable**
- Why?
  - Faster than lists
  - Protected from change
  - Can be used as keys in dictionaries, for instance a tuple of integers can represent coordinates in a sparse array.
  - Multiple return object from function
  - Swapping variable content `(a, b) = (b, a)`
  - Sequence unpacking `a, b, c = myTuple`
- `list(myTuple)` returns myTuple as a list. `tuple(myList)` does the opposite.

Sets

- **Set**. Contains unique objects in **no order** with **no identification**.
  - With a **list**, elements are ordered and identified by position. `myList[2]`
  - With a **dictionary**, elements are unordered but identified by some key. `myDict['myKey']`
  - With a **set**, elements stand for themselves. No indexing, no key-reference.
- Declaration: `fib=set( (1,1,2,3,5,8,13) )` returns the `set {1, 2, 3, 5, 8, 13}`.
- Supported operations: `len(s)`, `x in s`, `set1 < set2`, `union`, `intersection`, `add`, `remove`, `pop` ...

Boolean values and operators

- `True/False`
- `and`
- `or`
- `not`
- `a = True; b = False; a and b` [returns False].

Conditionals

if-elif-else statement

```
a =1
if a==1:
    print('a is one')
elif a==2:
    print('a is two)
else:
    print('a is not one or two')
```

- **Switch statements** via dictionaries if absolutely necessary.

While loops

```
a =10
while a>1:
   print('bigger than one')
   a = a - 1
else:
   print('smaller than one')
```

For loops

- **for loops can iterate over any iterable**.
- **iterables**: strings, lists, tuples, ranges, etc.

```
word = 'johan'
for letter in word:
   print(letter)
myList = ['']*10
for i in range(10):
   myList = 'johan' + str(i)
```

Comprehensions

- As in R, loops can be slow. For small loops executed many times to generate many results, try comprehensions:
- Set definition in mathematics

$$\{x \text{ for } x \in \mathcal{X}\}$$

where $\mathcal{X}$ is some a finite set.

$$\{f(x) \text{ for } x \in \mathcal{X}\}$$

- Comprehension types in Python:
  - `myList = [x for x in range(10)]`
  - `mySet = {sqrt(x) for x in range(10)}` (don't forget `from math import sqrt`)
  - `myDict = {x :  sqrt(x) for x in sample if x >= 0}`

Debugging

- pdb (https://github.com/spiside/pdb-tutorial)
- Debugging in Jupyter: Debugging in Jupyter: IPython.core.debug (https://davidhamann.de/2017/04/22/debugging-jupyter-notebooks/)

Misc

- Comments on individual lines starts with #
- Doc-strings can be used as comments spanning over multiple lines but this should be avoided `"""This is a looooong comment"""`