An introduction to Python

Andreas Bjerre-Nielsen

Agenda

1. Python: what it is; why and how we learn it
2. Python scripts and Jupyter
3. Fundamental data types
4. Basic Python procedures: operators, functions, methods etc.
5. Conditional logic, loops and reusable code

Introducing Python

Introduction

Why do we use Python?

- It is used everywhere - more examples here (https://www.python.org/about/success/)
  - High-tech manufacturing
  - Space shuttles
  - Large servers
  - Cutting-edge big data and data science.
- Leverages leading tools for machine learning and handling of big data.
- It has incredible tools for static and interactive visualization.

Introduction (2)

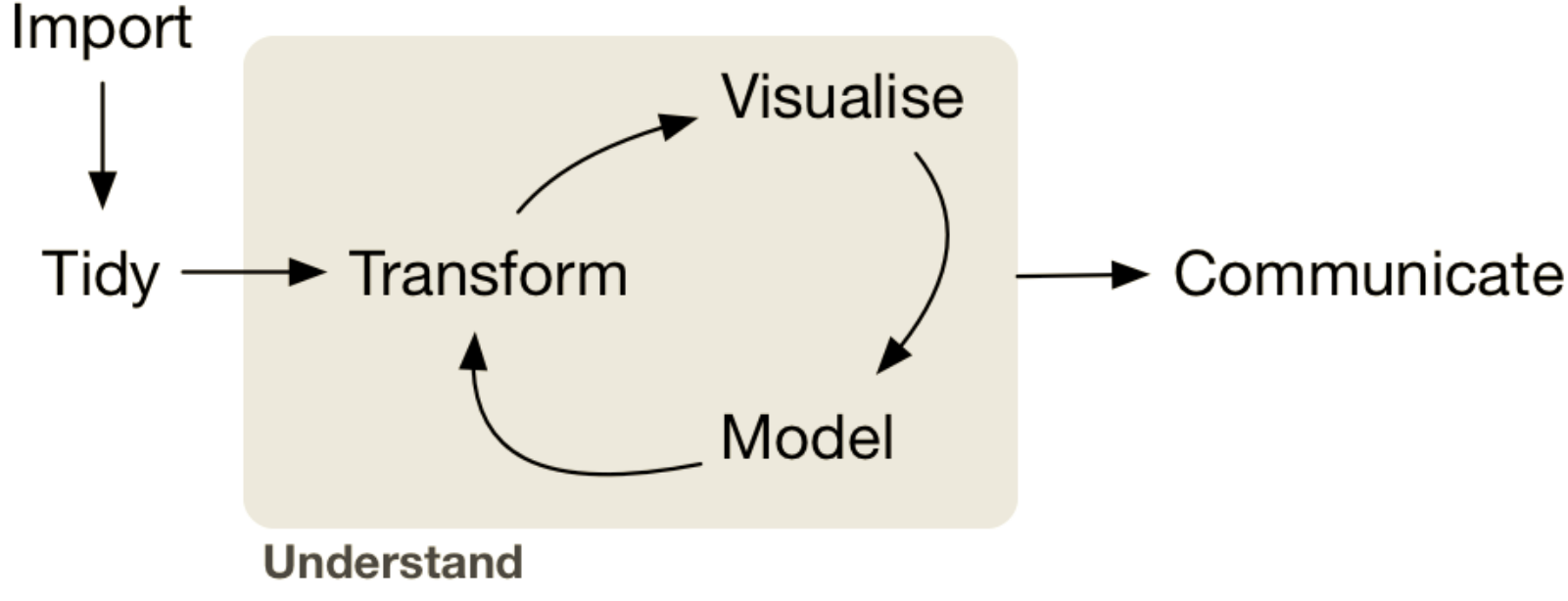
What is Python?

A general purpose programming language.

- Can do everything you can imagine a computer can do.
- E.g. manage databases, advanced computation, web etc.)

Introduction (3)

How do we use python for data science?



Help and advice

Learning how to code (I)

This course.. ain't easy..

Why go through this pain - personal experience.

Learning how to code (II)

Hadley Wickham

> *The bad news is that when ever you learn a new skill you're going to suck. It's going to be frustrating. The good news is that is typical and happens to everyone and it is only temporary. You can't go from knowing nothing to becoming an expert without going through a period of great frustration and great suckiness.*

Learning how to code (III)

Kosuke Imai

One can learn data analysis only by doing, not by reading.

Learning how to code (IV)

Practical advice

- Do not use the console, write scripts or preferably notebooks instead
- Be lazy: resuse code and write reusable code (functions)
- Think before you code
- Code is a medium of communication

    1. Between you and the computer
    2. Between you and other people (or future you)

Learning how to code (V)

How do we participate optimally?

- Have computer open
- Do everything we do on your own machine
  - Yes, typing it in, not copy-paste

Guide on getting help

Whenever you have a question you do as follows:

1: You ask your question to the person next to you.

2: You ask other people in your group.

3: You ask the neighboring group.

4: Either you raise your hand or you search on Google (more advice will follow).

The python shell and scripts

Python interpreter (1)

Shell access

The fundamental way of accessing Python is from your shell by typing *`python`*.

Everyone should be able to run the following commands and reproduce the output.

```
>>> print ('hello my friend')
hello my friend
```

```
>>> 4*5
20
```

Python interpreter (2)

Python scripts

The power of the interpreter is that it can be used to execute Python scripts.

What is a script?

These are programs containing code blocks.

Everyone should be able to make a text file called *test.py* in their current folder. The file should contain the following two lines:

```
print ('Line 1')
print ('Line 2')
```

Try executing the test file from the shell by typing:

python test.py

This should yield the following output:

```
Line 1
Line 2
```

The Jupyter framework

Jupyter (1)

Why Jupyter?

In our course we use it to:

- quickly load large datasets;
- apply Python's modules to compute statistics or visualizing figures
- use its many resources (e.g. create this slideshow)

What is Jupyter Notebook?

- Jupyter provides an interactive and visual platform for working with data.
- It is an abbreviation of Julia, Python, and R.

Jupyter (2)

How do we create a Jupyter Notebook?

We start Jupyter Notebook by typing `jupyter notebook` in the shell.

Try making a new notebook:

- click the button *New* in the upper right corner
- clicking on `Python 3`.

Jupyter (3)

How do we interact with Jupyter?

Jupyter works by having cells in which you can put code. The active cell has a colored bar next to it.

A cell is *`edit mode`* when there is a *green* bar to the left. To activate *`edit mode`* click on a cell.

A cell is in *`command mode`* when the bar to the left is *blue*.

Jupyter (4)

How do we add and execute cude?

Go into edit mode - add the following:

In [ ]:
```
A = 11
B = 15
A+B
```

Click the ► to run the code in the cell. What happens if we change A+B to A*B?

Jupyter (5)

How can we add cells to our notebook?

Try creating a new cell by clicking the **+** symbol.

Jupyter (6)

Most relevant keyboard short cuts

editing and executing cells

- enter edit mode: click inside the cell or press `ENTR`
- exit edit mode: click outside cell or press `ESC`.
- executing code within a cell is `SHFT+ENTR` or `CTRL+ENTR` (not same!)

adding removing cells (command mode only)

- delete a cell:d,d (press d twice)
- add cell: a above, b below

See all Jupyter keyboard shortcuts in menu (top): `Help > Keyboard Shortcuts`, or press `H` in command mode.

Jupyter (7)

What if I need more information?

Further resources can be found in the documentation and tutorial available here (http://jupyter.readthedocs.io/en/latest/).

Fundamental data types

Data types (1)

What are the most basic data types in Python?

Python supports many different data types. The four most basic are as follows.

- numbers come in two flavors
  - floating point numbers (**float**), e.g.: 3.14, 0.011
  - integers (**int**), e.g.: 1, 3, 8
- strings such as 'e', 'B', 'd' (**str**);
- boolean (**bool**) which is either `True` or `False`.

Data types (2)

We can store data as a variable X with one equal symbol, i.e. '='

Try creating a variable A which equals 1.3 by typing

In [ ]:
```
A = 4.5
```

Convert A to integer by typing:

In [ ]:
```
int(A)
```

We can do the same for converting to **float**, **str**, **bool**. Note some conversion are not allowed.

Data types (3)

What is an object in Python?

- A thing, anything - everything is an object.

Why use objects?

- Easy manipulable - has associated methods. Example of a float method:

In [ ]:
```
A.as_integer_ratio()
```

Print and debug

Basic printing

An essential procedure in Python is **print**.

Try executing some code:

In [ ]:
```
my_str = 'I can do anything in Python *!#'
print(my_str)
```

In [ ]:
```
my_str2 = 'Use the syntax \nto shift lines'
print(my_str2)
```

In [ ]:
```
my_var1 = 11
my_var2 = 45
print(my_var1, my_var2)
```

Why do we print?

Debugging

What happens if my code has errors?

Try executing the following code block:

In [ ]:
```
float('a')
```

Interpretation: The output message tells us that the string `'a'` cannot be converted to a floating number, which makes sense.

How do I handle it? Look hor help the standard way. Note when searching Google.

Operators and conditional logic

Operators (1)

What computations can python do?

An operator in Python manipulates various data types.

Have we seen any?

We have seen summation, i.e. **+**, which work for strings and numbers.

Other basic numeric operators:

- multiplication (*);
- subtraction (-);
- division (/);
- power, (**).

Operators (2)

What pitfalls exist for numerical division?

Division has two ways of usage. Try executing the following two code blocks

In [ ]:
```
print(5/2)
print(5//2)
```

What is the fundamental difference?

- / is "true" division, i.e. computes a floating number
- // is "floor" division, i.e. computes an integer by rounding down

To note: Associated with division is the modulus operator %; this computes the remainder from the integer division e.g. 3%2=1, 5%2=1, etc.

Operators (3)

Can we add a number to an existing?

Yes, this is done with the add-to-self operator **+=**.

Try example below:

In [ ]:
```
number = 8
number+= 2
number
```

Quiz: Does **-=**,*= work too? What about **+=** for strings?

Operators (4)

How can we test an expression in Python?

We can check the validity of a statement - using the equal operator, **==**, or not equal operator **!=**.

In [ ]:
```
3 == (2 + 1)
```

In [ ]:
```
21 == 4 * 5
```

In [ ]:
```
21 != 4 * 5
```

Operators (5)

How can we manipulate boolean values?

Combining boolean values can be done with using:

- the **and** operator - equivalent to **&**
- the **or** operator - equivalent to **|**

Let's try this!

In [ ]:
```
print(True | False,
      True & False)
```

A boolean value can be reversed done with the **not** operator.

In [ ]:
```
not (True and True)
```

Operators (6)

What other expressions are possible for numerical values?

Basic comparison of numbers include: **<**, **<=**, **>**, **>=**. Try it out:

In [ ]:
```
1 <= 2
```

Operators (7)

What are some operators for string data?

One string operation is to check whether a string S contains a substring T, this can be done with the **in** operator: S in T.

Try out:

In [ ]:
```
"Hello" in ("Goodbye")
```

In [ ]:
```
"Ye" in ("Yes")
```

Operators (8)

How can we access a given substring?

We access a substring in a string by `slicing` it. Let's try:

In [ ]:
```
my_str = 'abcde'
my_str[:2]
```

In [ ]:
```
start = 1
end = 3
my_str[start:end]
```

How does slicing work?

- By selecting from the start to end where start is included and end is excluded.*
- Python has zero-based indexing, see explanation here (https://softwareengineering.stackexchange.com/questions/110804/why-are-zero-based-arrays-the-norm).

Conditional logic

Executing code based on testing expressions.

Conditionals (1)

How can we activate code based on data?

A conditional execution of code, if a statement is true then active code.

In Python the syntax is easy with the **if** statement; when we have a boolean variable "statement" and a code block, simply

In [ ]:
```
if statement:
    code
```

Try this:

In [ ]:
```
if 4 == 4:
    print ('true!')
```

In [ ]:
```
my_statement = (5 == 4)
if my_statement:
    print ('true!')
```

Conditionals (2)

If the statement in our condition is false then we can execute other code with the **else** statement. This is done as follows:

In [ ]:
```
if statement:
    code
else:
    alternative code
```

Try running the following command.

In [ ]:
```
if (5 == 4):
    print ('true!')
else:
    print ('false..')
```

To note: Multiple conditions can be inserted with **elif** statement(s):

In [ ]:
```
if statement1:
    code1
elif statement2:
    code2
else:
    code_block_final
```

Conditionals (3)

Quiz: *How do we tell Python that we have a code block?*

By indenting the line with four whitespaces, example if A then B:

In [ ]:
```
if A:
    B
```

Containers

Lists (1)

How can we make a containter which is sequential and accessed by integers?

- One answer is the **list** data type. A list is a container for data. In terms of math it is an ordered array / vector.

A list named 'B' can contain the sequence of arbitrary elements 1,2,3,5. Try constructing this sequence as:

In [ ]:
```
B = [1,2,3,5]
```

Question: can a list mhave no elements?

Lists (2)

Do we know any methods that also work for lists?

Lists are similar to strings and share the **+**, **in** operator and `slicing` method.

Quiz: How did these work for strings?

- The **in** operator can be used as A in B where it returns True if there is A is an element in the array B, otherwise False.

Lists (3)

Exercise: Try checking if the integer 2 is in B. Try also checking 4.

A possible solution:

In [ ]:
```
B = [1,2,3,5]
check2 = 2 in B
check4 = 4 in B
print(check2, check4)
```

Slicing works exactly as it does for strings by returning a subsequence: B[2:] returns [1,2] and B[1:3] returns [2,3].

Lists (4)

What else can lists be used for?

We can change the elements in a list. Substitution of an element in the list can be done by slicing. Try this:

In [ ]:
```
L = [1,2,1]
L[1] = 0
print(L)
```

Adding elements to a list can be done as L.append(O) where L is our list and O is the object we add.

To note: more methods for lists can be seen here (https://docs.python.org/3/tutorial/datastructures.html#more-on-lists), including removal of elements with **pop** and **remove**.

Dictionaries

How can we make a container which is accessed by arbitrary keys?

By using a dictionary (**dict**). Try executing the code below:

In [ ]:
```
my_dict = {'David':'Economist', 'Andreas':'Economist', 'Snorre':'Sociologist'}
print(my_dict['Snorre'])
```

Dictionaries can be constructed from two associated lists. These are tied together with the **zip** function. Try the following code:

In [ ]:
```
keys = ['a','b']
values = [1,2]
key_value_pairs = zip(keys, values)

my_dict2 = dict(key_value_pairs)
print(my_dict2['b'])
```

Loops

A `loop` is to run a repeated process.

For loops

Why are lists so powerful?

Lists can be used to iterate over its elements - this created a *finite* loop, called the **for** loop.

Example - try the following code:

In [ ]:
```
A = []
B = [1,2,3,5]

for i in B:
    i_squared = i**2
    A.append(i_squared)

print(A)
```

For loops are smart when: iterating over files in a directory; iterating over specific set of columns.

How does Python know where the code associated with inside of the loop begins?

While loops

Can we make a loop without specifying the end?

Yes, this is called a **while** loop. Example - try the following code:

In [ ]:
```
i = 0
L = []
while (i<3): # i<3 is condition for while loop to keep going
    L.append(i*2) # add i times 2 to L
    i += 1 # add 1 to i
print(L)
```

Why is this smart?

- Can be applied in scraping, make processes that keeps running, model which converges, etc.

Reusable code

Why do we reuse code?

- To save time.

Functions

What procedures have we seen?

How can we make a reusable procedure?

- We make a Python function with the **def** syntax. Try this:

In [1]:
```
def squared_plus_1(n): # takes input n
    n_squared = n**2 # n squared
    return n_squared+1

squared_plus_1(3.1)
```

Out[1]:
```
10.610000000000001
```

Functions (2)

Exercise: How can we define a function that takes cubic (power 3) and subtracts 2?

A solution could be as follows:

In [3]:
```
def my_fct(x):
    return x**3 - 2
```

Let's test this

In [4]:
```
my_fct(2)==6
```

Out[4]:
```
True
```

Functions (3)

We can make functions that allows multiple arguments:

In [ ]:
```
def my_subtract(arg1, arg2):
    return arg1-arg2

my_subtract(7, 5)
```

We can call the functions with keyword arguments - this allows us to use any order. Example:

In [ ]:
```
my_subtract(arg2=7, arg1=5)
```

Functions (4)

Functions can be created to allow optional arguments. This is done by setting default values. Example:

In [6]:
```
def multiply3(arg1, arg2, arg3=1):
    return arg1*arg2*arg3

multiply3(3,2), multiply3(3,2,2)
```

Out[6]:
```
(6, 12)
```

Modules

How can we load resources made by others in Python?

We load a module. Try importing pandas:

In [8]:
```
import pandas as pd
```

Let's create a Pandas DataFrame.

In [11]:
```
row1 = [1,2]
row2 = [3,4]
table = [row1, row2]

df = pd.DataFrame(table, columns=['A', 'B'])

print(df)
```

```
   A  B
0  1  2
1  3  4
```

Modules (2)

What is a Pandas DataFrame?

A matrix where rows and columns have names.

Objects can have useful methods, i.e. functions.

Example, in a DataFrame we can access the column called 'col1':

In [40]:
```
print(df.sort_values(['col2'], ascending=False))
```

```
   col1  col2
1     3     4
0     1     2
```

Modules (3)

Let's try loading another module.

In [ ]:
```
import seaborn as sns
```

- Is seaborn installed?

Final remarks

Summary

In this lecture we learned how to use:

- Python scripts and Jupyter
- Fundamental data types: numeric, string and boolean
- Operators: numerical, strings and logical
- Conditional logic
- Containers: lists, dictionaries
- Loops: for and while
- Reuseable code: functions and modules

More useful material

Due to the scope of the course we cannot cover everything. Below is a list of Python skills that may be useful.

- Opening and closing files (http://www.python-course.eu/python3_file_management.php).
- List comprehension (http://www.python-course.eu/python3_list_comprehension.php) for fast to make one-line for loops.
- Datetime (https://docs.python.org/3/library/datetime.html) to handle time data (we will cover this briefly with Pandas).
- String formatting (http://www.python-course.eu/python3_formatted_output.php) to improve your skills of printing output
- Regular expression (http://www.python-course.eu/python3_re.php) (we will cover this with scraping)