Python Workshop

Social Data Science, summer 2019

Kristian Urup Olesen Larsen[1] Jakob Jul Elben[2]

July 18, 2019

[1] CEBI, University of Copenhagen

[2] DATAMAGA

If you have not yet installed python, now is the time to do it.

Get it via anaconda: https://www.anaconda.com/

How to learn python?

1. Ask questions to your code that help you understand what you want it to do.
2. Ask your questions to google, they know the answer.
3. Read the error messages, they are there for a reason.
4. Google the error messages. Others have the same problems as you.
5. Read the documentation (but remember it is technical docs, not a novel).
6. If your code is broken try to change *something*. If it works, good. Otherwise repeat.
7. If this doesn't work, google some more.

The Zen of Python

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
# ...
Readability counts.
# ...
In the face of ambiguity, refuse the temptation to guess.
# ...
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Data types

Elementary data types

- integers $(\ldots, -2, -1, 0, 1, 2, \ldots)$

```
>>> x = 5
>>> type(x)
int
```

- floats $(\approx \mathbb{R})$

```
>>> x = 5.3
>>> type(x)
float
```

- strings (Letters and other symbols)

```
>>> x = 'Hello'
>>> type(x)
str
```

- booleans (0/1):

```
>>> x = True
>>> type(x)
bool
```

Composite data types

- tuples (immutable ”vectors”)

```
>>> x = (2, -5.3, 'a')
>>> type(x)
tuple
```

- lists (mutable ”vectors”)

```
>>> x = [2, -5.3, 'a']
>>> type(x)
list
```

- dictionaries (key-value data)

```
>>> x = {'height': 5, 'width': 2}
>>> type(x)
dict
```

- sets (lists, but doesn’t allow duplicates)

```
>>> x = set([2, -5.3, 'a', 2])
>>> type(x)
set
>>> x
{2, -5.3, 'a'}
```

Working with: numbers and strings

- You can do arithmetic with numbers

```
>>> 2 + 3.0     # also try -, *, /, **
5.0
>>> 7 % 3       # modulo
1
```

- as well as with strings

```
>>> 'Hello' + ' world'
'Hello world'
>>> 'Hello' * 3
'HelloHelloHello'
```

- Strings can be ”sliced” to get substrings

```
>>> s = 'This is a sentence'
>>> s[0]
'T'
>>> s[0:4]
'This'
>>> s[-1]
'e'
```

Working with: tuples and lists

- slicing works as with strings

```
>>> l = [1, 1, 2, 3, 5, 8, 12]
>>> l[-1]
12
```

- Tuples cannot be changed after they have been created, lists can

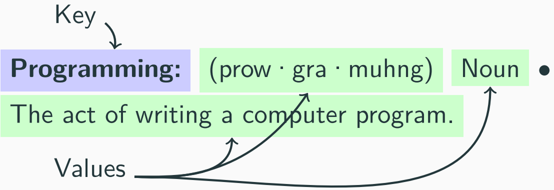
```
>>> l.append(20)     # doesn't work with a tuple
>>> l
[1, 1, 2, 3, 5, 8, 12, 20]
```

- Both tuples and lists can contain (almost) anything

```
>>> a = []                         # an empty list
>>> b = (-1,-2,-3)                 # a tuple
>>> c = (1, a, 'a', [2, 3, b])     # a and b are placed in c
>>> c
(1, [], 'a', [2, 3, (-1, -2, -3)])
```

Working with: dictionaries

- dicts are not like lists, tuples and sets. They are key-value structured. This is similar to an actual dictionary:



- In python with a dictionary:

```
>>> my_dict = { 'Programming' :
  ['(prow·gra·muhng)',
  'Noun',
  'The act of writing a computer program.']
        }
```

Working with: dictionaries

- dicts are accessed by their keys using [ ] notation

```
>>> my_dict['programming']
['(prow·gra·muhng)', ... ]
```

- You can get a list-like object of all the keys/values by

```
my_dict.keys()
my_dict.values()
```

Control flow

boolean values

- bools in python can be `True` and `False`.
- they are the result of *logical statements*, involving $\geqslant$, $\leqslant$, $>$, $<$, $==$, $\neq$ operators.

```
>>> color = 'red'                   # setup
>>> number = 5                      # setup

>>> color == 'black'
False

>>> color != 'blue'
True

>>> number > 2
True
```

if/elif/else

- You can control the flow of your program using `if`, `elif` and `else` when you need to run code conditionally

```
>>> color = 'blue'             # setup
>>>
>>> if color == 'black':
>>>     print('The color is black')
>>> elif color == 'blue':
>>>     print('The color is blue')
>>> else:
>>>     print('The color is not blue or black')
```

try/except/finally

- If your code *might* fail, you can `try` it. If it fails the `except` block is run.

```
>>> try:
>>>     5 + 's'
>>> except:
>>>     print('could not add 5 and s')
>>> finally:          # the finally block can be omitted
>>>     print('this part is always executed')
```

for loops

- The `for` loop goes through each element of something, one at a time

```
>>> something = ['h', 'e', 'l', 'l', 'o']
>>>
>>> for letter in something:
>>>     print(letter)
>>>
>>> for i in range(10):
>>>     print(i**2)
```

- This works with any object you think it should work with (any iterable).

while loops

- The `while` loop keeps running as long as a condition is `True`

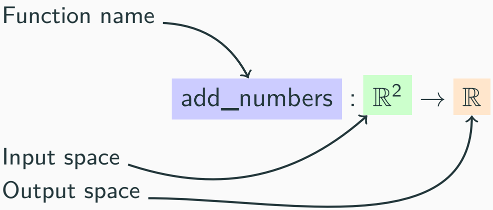
```
>>> my_number = 0
>>>
>>> while my_number < 10:
>>>     print('my number is ', my_number)
>>>     my_number = my_number + 1
```

- Of course something in the loop body must change the status of the logical condition, otherwise the loop runs forever.

Functions

Functions

- Functions are ”recipes” for carrying out specific operations on data.
- Completely analogous to mathematical functions



$$\text{add\_numbers} : \mathbb{R}^2 \to \mathbb{R}$$

- Define a function in python with the `def` keyword:

```
>>> def add_numbers(x1, x2):
>>>     result = x1 + x2
>>>     return result
```

Functions

- Another example, let $\mathbb{S}$ be the set of strings, and $\mathbb{F}$ the set of valid python functions

$$g : \mathbb{S} \times \mathbb{F} \to \mathbb{R}$$

$$g(s, f) = \begin{cases} f(0) \text{ if } s = \text{'zero'} \\ f(1) \text{ else} \end{cases}$$

- In python:

```
>>> def g(s, f):
>>>     if s == 'zero':
>>>         return f(0)
>>>     else:
>>>         return f(1)
```

- Functions can work with (almost) any object you can define in python.

Functions

- Functions don't do any computation when defined. To ”use” a function it must be *called*

```
>>> add_numbers(5,8)
13
```

- Second example:

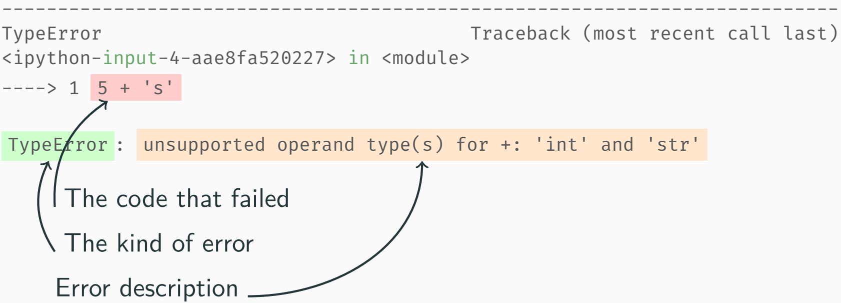
```
>>> def plus_one(x):
>>>     return x + 1

>>> g('zero', plus_one)
1
>>> g('whatever', plus_one)
2
```

Various bits

Errors

- Error messages (tracebacks) contain useful information

```
>>> 5 + 's'
```



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-aae8fa520227> in <module>
----> 1 5 + 's'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The code that failed

The kind of error

Error description

packages

- While python claims to be ”batteries included” you will need to install a bunch of external packages for data science.
- For this you can use either `pip` or `conda` (prefered), in the *TERMINAL* run

```
$ conda install numpy
# or
$ pip install numpy
```

- after installing a package you can import it in python

```
>>> import numpy          # or
>>> import numpy as np    # and/or
>>> from numpy import array
```

Mistakes often made

Mistakes often made

- Python is case sensitive, misspelling keywords (e.g. `For, In, While, If`) raises `SyntaxError: invalid syntax`
- Indentation matters! All indented blocks are marked with an (easily forgotten) colon, e.g.

```
# correct
for x in range(10):
    print(x)
# wrong, raises SyntaxError
for x in range(10)
    print(x)
```

- Make sure that parentheses match. (also raises `SyntaxError`)
- Remember commas where required. (`SyntaxError` as well)