2015 Computational Social Science Workshop

Day 1 - Introduction to `python` - Part 1 / 3

with **Jongbin Jung** (jongbin at stanford.edu)

- PhD student. Decision analysis, MS&E

All material for days 1 (intro to `python`) and 2 (web scraping with `python`) publicly available here (https://github.com/jongbinjung/css-python-workshop)

Meet `python`

Why `python`?

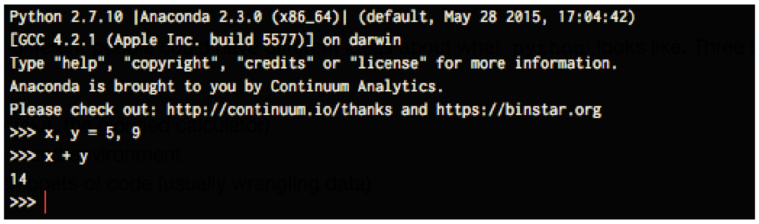
- fast
- modular
- object oriented
- tons of libraries
- (relatively) easy to read
- *gets the job done*

Choice of primary programming language should really depend on your preferred style of thinking and tools of choice (e.g., linear models? decision trees?). But you'll never know if it's right (or wrong) for you unless you give it a try!

What do you mean by `python`?

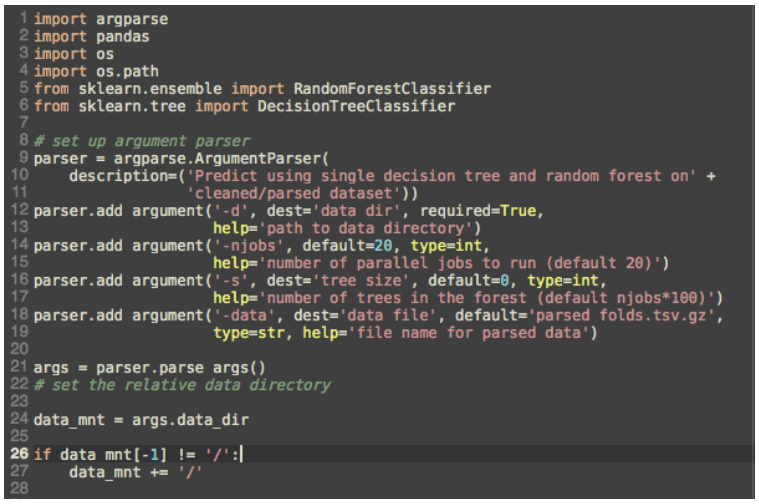
`Python` is the name of a programming language. Period. But different people could have different ideas about what '`python`' looks like. Three broad categories *I* have in mind are:

1. Running interactive commands in the `python` interpreter (aka, the glorified calculator)



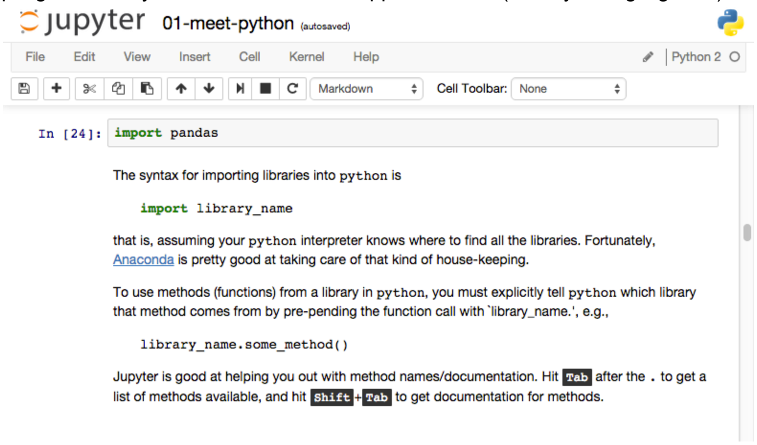
```
Python 2.7.10 |Anaconda 2.3.0 (x86_64)| (default, May 28 2015, 17:04:42)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
>>> x, y = 5, 9
>>> x + y
14
>>> 
```

2. `Python` development in some kind of text editor or specialized environment



```python
import argparse
import pandas
import os
import os.path
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier

# set up argument parser
parser = argparse.ArgumentParser(
    description=('Predict using single decision tree and random forest on' +
                 'cleaned/parsed dataset'))
parser.add_argument('-d', dest='data_dir', required=True,
                    help='path to data directory')
parser.add_argument('-njobs', default=20, type=int,
                    help='number of parallel jobs to run (default 20)')
parser.add_argument('-s', dest='tree_size', default=0, type=int,
                    help='number of trees in the forest (default njobs*100)')
parser.add_argument('-data', dest='data_file', default='parsed_folds.tsv.gz',
                    type=str, help='file name for parsed data')

args = parser.parse_args()
# set the relative data directory

data_mnt = args.data_dir

if data_mnt[-1] != '/':
    data_mnt += '/'

```

3. Research-type scripting with heavy documentation and snippets of code (usually wrangling data)



```
In [24]: import pandas
```

The syntax for importing libraries into `python` is

```
import library_name
```

that is, assuming your `python` interpreter knows where to find all the libraries. Fortunately, Anaconda is pretty good at taking care of that kind of house-keeping.

To use methods (functions) from a library in `python`, you must explicitly tell `python` which library that method comes from by pre-pending the function call with `library_name.', e.g.,

```
library_name.some_method()
```

Jupyter is good at helping you out with method names/documentation. Hit `Tab` after the `.` to get a list of methods available, and hit `Shift`+`Tab` to get documentation for methods.

For the purpose of a proper "`python` introduction", we'll look at all three flavors, and try to spend more time on the workflow that participants find most relevant to their work/research.

(Some of the material for Parts 1 and 2 were adapted from The Python Tutorial (https://docs.python.org/2/tutorial/))

Anaconda

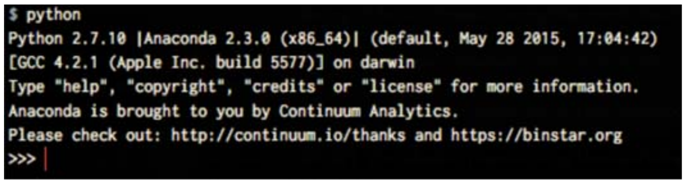
- **Anaconda (https://store.continuum.io/cshop/anaconda/)** is one of many `python` distributions
- it's free (some distributions aren't).
- comes pre-installed with a lot of useful libraries
- support many different workflows out-of-the-box
- think of it as a **"`python` starter kit"**
- as far as I've seen, it's the most painless way to get things up-and-running

- download/install from here (https://store.continuum.io/cshop/anaconda/)

1. `python` interpreter (aka glorified calculator)

You can start using `python` right away, by running the `python` interpreter in the command line (Windows) or terminal (OS X / *nix)

- **Windows**
  - `Win + r` , type `python` and hit `OK` (or `Enter`)
- **OS X / *nix**
  - Open a terminal
  - type `python` and hit `Enter`



```
$ python
Python 2.7.10 |Anaconda 2.3.0 (x86_64)| (default, May 28 2015, 17:04:42)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
>>>
```

Numbers

Most of your numerical needs will fall into two types: integers (1, 2, ...) and floating point (0.5, 1.75, ...). Let's start with some basic calculations:

```
In [1]: 296719 + 445567  # add stuff

Out[1]: 742286
```

```
In [2]: 762301 - 162379  # subtract stuff

Out[2]: 599922
```

```
In [3]: 148501 * 220131  # multiply stuff

Out[3]: 32689673631L
```

```
In [4]: 1 / 10  # divide stuff ...?

Out[4]: 0
```

You might notice two things:

1. Anything after the # is ignored. These are called **comments**. It's good practice to use them to leave notes for *future* you.
2. 1/10 is **not** 0.

What's happening?

`Python` never asks you what type each number/variable is. It assumes the type of a variable based on its value, in this case 1 and 10 are both assumed to be integers. And the division operation (`/`) determines the type of its result based on the type of the operands. If both operands are integers, *floor division* is performed and an integer is returned. If either operand is a float, classic division is performed and a float is returned.

So, to *implicitly* tell `python` that you're expecting a floating point number, you have to make one (or both) of the numbers look like a floating point number:

```
In [5]: 1 / 10.0
Out[5]: 0.1
```

This is an important thing to know about `python`, especially if you're used to different programming languages (statically typed languages such as C, Java, ...)

`Python` will *not* keep track of types for you.

It is absolutely up to you to make sure that your variables are the right type.

Some more useful operators are:

```
In [6]: 5.0 // 2  # an explicit floor division
Out[6]: 2.0

In [7]: 5.0 % 2  # modulo operation (finds the remainder after division)
Out[7]: 1.0

In [8]: 2 ** 10  # calculate powers
Out[8]: 1024
```

You can assign (store) values to variables with an equal (=) sign. Again, since `python` isn't interested in knowing the type of each variable before runtime, you don't need to declare any. Many people love or hate `python` for this.

```
In [9]: sales = 1000
        costs = 1300
        sales - costs
Out[9]: -300
```

Note that the interpreter doesn't return anything when you assign/define variables. Also, trying to use variables without defining them will result in an error.

```
In [10]: profit
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-10-65280be1bd6e> in <module>()
----> 1 profit

NameError: name 'profit' is not defined
```

There are more types (e.g., long - for huge integers, complex numbers, fractions) if you think you'll need them. See the docs (https://docs.python.org/2/library/types.html) for a comprehensive documentation of `python`'s built-in types.

Strings

One of the many things that `python` is *really* good for is manipulating strings. Strings are enclosed in either single quotes(`'...'`) or double quotes (`"..."`). The only difference betwen the two is that you need to escape literal double quotes with `\` if you're using single quotes, and vice versa. Examples,

```
In [11]: 'This is a string in single quotes, and it works fine'

Out[11]: 'This is a string in single quotes, and it works fine'
```

```
In [12]: 'This is a string in single quotes, but it's broken!'
```

```
  File "<ipython-input-12-191c80bf77a1>", line 1
    'This is a string in single quotes, but it's broken!'
                                               ^
SyntaxError: invalid syntax
```

```
In [13]: 'This is a string with single quotes, and now it\'s fixed!'

Out[13]: "This is a string with single quotes, and now it's fixed!"
```

```
In [14]: "So, there's incentive to use double quotes"

Out[14]: "So, there's incentive to use double quotes"
```

That should make it pretty clear. Note that when escaping a single quote in a string enclosed in single quotes, the interpreter internally changes the enclosing quotes to double quotes. The two are absolutely identical, so you should use whichever set of quotes you prefer to enclose your strings, as long as you're consistent.

Use the `print` statement to make output more readable by omitting the enclosing quotes and printing special characters escaped with `\`.

```
In [15]: 'enclosing double quotes ("") and single quotes(\'\') are the same thing in python'

Out[15]: 'enclosing double quotes ("") and single quotes(\'\') are the same thing in python
         '
```

```
In [16]: print 'enclosing double quotes ("") and single quotes(\'\') are the same thing in py
         thon'
```

```
enclosing double quotes ("") and single quotes('') are the same thing in python
```

Stings can also have more than one line. You can either use an explicit line-break character (`\n`):

```
In [17]: print 'This string has\ntwo lines!'
```

```
This string has
two lines!
```

... or use triple quotes `'''...'''` or `"""..."""`

```
In [18]: print '''
         This string has
         two lines!
         '''
```

```

This string has
two lines!

```

If you look carefully enough, you'll notice that the last string actually has four lines. This is because the triple quotes literally encode all white spaces, including the new lines after the first `'''` and the last `"!"`. To avoid this, you can escape the new lines with `\`

```
In [19]: print '''\
         This string (really) has
         two lines!\
         '''
```

```
This string (really) has
two lines!
```

Unlike some other stingy languages, the plus operator (`+`) does exactly what you'd expect it to do with strings!

```
In [20]: first_name = 'Jongbin'
         last_name = 'Jung'
         full_name = first_name + ' ' + last_name
         print 'Hello,', full_name
```

```
Hello, Jongbin Jung
```

Even the multiplying operator (`*`) works!

```
In [21]: print 'Sing, ' + 'la ' * 3
```

```
Sing, la la la
```

Strings, like many things in `python`, can be indexed (subscripted). The first element (character) has index 0.

```
In [22]: job = 'jedi'
         job[0]  # character at position 0
```

```
Out[22]: 'j'
```

`Python` will yell at you if you go out of range.

```
In [23]: job[10]

         ---------------------------------------------------------------------------
         IndexError                                Traceback (most recent call last)
         <ipython-input-23-72005d25ec5c> in <module>()
         ----> 1 job[10]

         IndexError: string index out of range
```

But you *can* go backwards with a negative index, -1 being the right-most character.

```
In [24]: job[-1]  # right-most character

Out[24]: 'i'
```

Slicing is another useful way to get subsets of your string.

```
In [25]: job[0:2]  # characters from position 0 (included) to 2 (excluded)

Out[25]: 'je'

In [26]: job[2:4]  # characters from position 2 (included) to 3 (excluded)

Out[26]: 'di'
```

Omitting the first slice index will default to zero (the first element)

```
In [27]: job[:3]  # slice from the first element (included) to position 3

Out[27]: 'jed'
```

Omitting the second index will default to the size of the string, which includes the last element.

```
In [28]: job[1:]  # slice from position 1 (included) to the end

Out[28]: 'edi'
```

Use slices creatively, to make your life easier!

```
In [29]: job[-2:]  # slice the last two characters

Out[29]: 'di'
```

Unlike indexing, slicing is generous to ambitious ranges

```
In [30]: job[0:100]

Out[30]: 'jedi'
```

Strings in `python` are `immutable`, meaning they can't be changed. In other words, you can't assign a value to a string index.

```
In [31]: job[0] = 'T'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-d4729ccf68c2> in <module>()
----> 1 job[0] = 'T'

TypeError: 'str' object does not support item assignment
```

Instead, you have to build a new string from the existing string.

```
In [32]: job = 'T' + job[1:]
         print job
```

```
Tedi
```

The built-in `len()` function will return the length of a string.

```
In [33]: len(job)

Out[33]: 4
```

Lists

Lists are perhaps the most versatile array-like data types in `python`. A list is declared (and printed) as a comma-separated value between square brackets.

```
In [34]: names = ['Luke', 'Leia', 'Han']
         print names
```

```
['Luke', 'Leia', 'Han']
```

```
In [35]: numbers = [45, 8, 21, 30]
         print numbers
```

```
[45, 8, 21, 30]
```

```
In [36]: mixed = [3, 4, 'five', 'six']
         print mixed
```

```
[3, 4, 'five', 'six']
```

Lists (and in fact, all built-in `python` sequence (https://docs.python.org/2/glossary.html#term-sequence) types) can be indexed and sliced.

```
In [37]: names[0]  # returns the item at position 0

Out[37]: 'Luke'
```

```
In [38]: mixed[1:3]  # slicing returns a new list

Out[38]: [4, 'five']
```

Also like strings, the + operation concatenates two lists

```
In [39]: names + ['Anakin', 'Padme']

Out[39]: ['Luke', 'Leia', 'Han', 'Anakin', 'Padme']
```

Unlike strings, lists are mutable, which means you can change item values, even to different types.

```
In [40]: mixed[0] = 30  # change indexed item
         print mixed

         [30, 4, 'five', 'six']
```

You can use the `append()` method to add new items to a list. Methods of an object are called directly on the object with a dot (`.`)

```
In [41]: mixed.append('seven')
         print mixed

         [30, 4, 'five', 'six', 'seven']
```

You can also assign values to slices of a list.

```
In [42]: mixed[0:2] = ['three', 'four']  # change sliced list items
         print mixed

         ['three', 'four', 'five', 'six', 'seven']
```

Because assignment to a slice replaces a subset of the list, it can be used to delete items.

```
In [43]: mixed[2:] = []  # delete items from position 2 to the end of the list
         print mixed

         ['three', 'four']
```

The built-in `len()` function also works as it would with strings.

```
In [44]: len(names)

Out[44]: 3

In [45]: len(numbers)

Out[45]: 4

In [46]: len(mixed)

Out[46]: 2
```

More on Strings and Lists

Both strings and lists have a bunch of helpful mehtods.

See the docs here (https://docs.python.org/2/library/stdtypes.html#string-methods) for strings and here (https://docs.python.org/2/tutorial/datastructures.html#more-on-lists) for lists.

Some of the more often used methods are illustrated below:

```
In [47]: s = 'Hi, my name is Jongbin. Yes, Jongbin.'
         s.split()  # splits the string into a list (split at spaces by default)

Out[47]: ['Hi,', 'my', 'name', 'is', 'Jongbin.', 'Yes,', 'Jongbin.']
```

```
In [48]: s.split(',')  # can specify which character to split at

Out[48]: ['Hi', ' my name is Jongbin. Yes', ' Jongbin.']
```

```
In [49]: s.count('n')  # count the number of non-overlapping occurrences of a substring

Out[49]: 5
```

```
In [50]: s.count('Jongbin')

Out[50]: 2
```

```
In [51]: s.upper()  # makes everything uppercase

Out[51]: 'HI, MY NAME IS JONGBIN. YES, JONGBIN.'
```

```
In [52]: s.lower()  # makes everythin lowercase

Out[52]: 'hi, my name is jongbin. yes, jongbin.'
```

```
In [53]: s.lower().count('y')  # methods that return a string can be chained

Out[53]: 2
```

```
In [54]: ':'.join(s.split())

Out[54]: 'Hi,:my:name:is:Jongbin.:Yes,:Jongbin.'
```

That last one is a little tricky. So, `str.join(some_sequence)` will take each item of `some_sequence` and stick them together with the value of `str` inbetween, making a single large string. It may seem like a crazy thing to do, but is actually pretty useful whem converting data into comma-separated values. i.e.,

```
In [55]: ','.join(['some','data','in','a','list'])

Out[55]: 'some,data,in,a,list'
```

And also some tricks with lists:

```
In [56]: fruits = ['mango', 'apple', 'banana', 'apple', 'kiwi', 'mango']
         fruits.append('orange')  # add an item to the end of the list
         print fruits

         ['mango', 'apple', 'banana', 'apple', 'kiwi', 'mango', 'orange']
```

```
In [57]: fruits.remove('apple')  # remove the first item in the list that matches the argumen
         t
         print fruits

         ['mango', 'banana', 'apple', 'kiwi', 'mango', 'orange']
```

```
In [58]: fruits.index('mango')  # return the index of the first item matching the argument

Out[58]: 0
```

```
In [59]: fruits.count('mango')  # return the number of times x appears in the list

Out[59]: 2
```

```
In [60]: eat = fruits.pop(0)  # return and remove item at position 0 from list (removes last
         item if no index is specified)
         print fruits

         ['banana', 'apple', 'kiwi', 'mango', 'orange']
```

```
In [61]: print eat  # the item previously at position 0 ('mango') is now 'popped' into the va
         riable eat

         mango
```

Not exactly a method, but the `in` keyword is useful for checking if a list contains a particular item.

```
In [62]: 'banana' in fruits

Out[62]: True
```

```
In [63]: 'tomato' in fruits

Out[63]: False
```

You can still do much more with `python`, even in the interpreter. But anything beyond a single line starts getting tedious in the interpreter, so now might be a good time to move on to the next stage: **`python` development**.

Exercise 1.

Before moving on, just to make sure that everyone's on the same page, let's try a few tasks:

1. declare a string variable named **`s`**, that has the value

   > "double quotes" and single 'quotes' are equally acceptable in python

2. make `python` count how many times the letter `t` appears in the string **`s`**
3. replace all quotation marks in the string **`s`** with an underbar ('_')
   - to be precise, you're not *replacing* the quotations, but *reassigning* the variable **`s`** with a copy of the old **`s`** that has underbars replacing quotations; remember that `python` strings are **immutable** (i.e., they are NEVER changed, only reassigned)
   - this was not covered above, but you should use `python`'s `replace()` method for strings; now would be a good time to practice reading the docs (https://docs.python.org/2/library/stdtypes.html#str.replace (https://docs.python.org/2/library/stdtypes.html#str.replace))
4. split the string **`s`** into a list named **`words`**
5. count the length of the string **`s`** and the list **`words`**
6. join the elements of the list **`words`** into a comma separated string