

# Python crash course

## Basic python programming

with Jongbin Jung

All material publicly available [here \(https://github.com/jongbinjung/mse231-workshop\)](https://github.com/jongbinjung/mse231-workshop)

### 2. python programming - 1 of 3

Lines of python code can be saved to a plain text file, conventionally appended with a `.py` extension, which can be read by the interpreter to run a python program. A common setup for python development is to have

1. a text editor of choice
2. a command line/terminal open to run the `.py` file
3. (optionally) a interpreter for testing small pieces of code

Having your own setup can be great if you have a favorite text editor and love pushing commands around different windows. Some text editors have pretty good support for python development, too.

Another option is to use an IDE (Integrated **D**evelopment **E**nvironment), specifically catered to your python development needs. [PyCharm \(https://www.jetbrains.com/pycharm/\)](https://www.jetbrains.com/pycharm/) is a pretty one, and will be familiar if you've worked with IDEA, Android Studio, WebStorm, PhpStorm, etc. (They're all based on the same platform.) Today, we'll be working with spyder because

1. it's free
2. it comes included with Anaconda
3. it loads faster (compared to PyCharm)

## Launching spyder

Spyder is best launched from Anaconda's launcher.

- launch the launcher
  - **Windows**
    - `Win` + `r`, type launcher and hit `OK` (or `Enter`) <img src=img/ss\_win\_run\_launcher.png alt="Run command on Windows" width=300>
  - **OS X / \*nix**
    - Open a terminal
    - type launcher and hit `Enter` <img src=img/ss\_osx\_term\_launcher.png alt="Run launcher from a terminal">

<img src=img/ss\_launcher.png alt="Anaconda launcher" width=600>

- hit `Launch` for spyder-app

<img src=img/ss\_spyder.png alt="Jupyter open in a browser" width=600>

By default, spyder has a text editor on the left pane, an interactive console (to which you can also send selected commands from the text editor pane with `Shift` + `Enter`) and object/variable/file browsers on the right side.

Feel free to explore and get used to the spyder environment before we move on.

(One feature I find particularly useful is the `Ctrl` + `i` shortcut, which displays documentation for the object at my current cursor, whether in the editor or console.)

## if statements

An example should suffice

```
In [ ]: x = int(raw_input('Give me a BIG number: '))
        if x < 0:
            print 'You\'re joking, right?'
        elif x < 1e3:
            print 'Try harder ... '
        else:
            print 'Nice.'
```

Some notes on the above code:

- the `raw_input()` function (as you've now seen), prompts the user for an input
- the `int()` (tries to) convert string values to integers (`raw_input()` will always return the user's input as a string)
- `elif` is short for `else, if`, and there can be none or more than one `elif` sequences
- the `else` clause is optional

One more thing that's implicit but **extremely** important: **Indents**.

- python, unlike many other languages out there, doesn't use curly brackets `{}`
- instead, blocks of grouped code are identified by the level of indents (this is something to get used to, if you've never seen it before)
- word of caution: NEVER USE `Tab` (don't worry, spyder changes all your `Tab`s to four spaces by default, which is the [PEP 8 spec for indentation \(https://www.python.org/dev/peps/pep-0008/#indentation\)](https://www.python.org/dev/peps/pep-0008/#indentation) in python)

## for statements

The `for` statement in python iterates over the items of any sequence (e.g., lists and even strings!), in the order that they appear in the sequence.

```
In [1]: names = ['Jamie', 'Cersei', 'Jon', 'Sansa']

for name in names:
    print name, 'has', len(name), 'characters and starts with a', name[0]
```

```
Jamie has 5 characters and starts with a J
Cersei has 6 characters and starts with a C
Jon has 3 characters and starts with a J
Sansa has 5 characters and starts with a S
```

The example above introduces a few new concepts:

- the variable `name` is defined along with the declaration of the `for` statement. It doesn't need to exist beforehand
- the `print` statement can take multiple expressions of different types, (try to) change them to a string, and insert a space between each item (separated by commas)
- it's good practice to use plurals for collections (`names` for the list) and singulars for individual items (`name` for each name)

You can also loop over a string, one character at a time.

```
In [2]: vowels = ['a', 'e', 'i', 'o', 'u'] # make a list of vowels
        for name in names:
            vowel_count = 0 # initialize the vowel count
            for char in name:
                if char in vowels:
                    vowel_count += 1
            print name, 'has', vowel_count, 'vowel(s)'
```

```
Jamie has 3 vowel(s)
Cersei has 3 vowel(s)
Jon has 1 vowel(s)
Sansa has 2 vowel(s)
```

You can use the built-in `range()` function to do a more 'classic' for loop over a sequence of numbers.

```
In [3]: range(10)
```

```
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`range(len)` generates the legal indices (starting from 0) for a sequence of length `len`. You can also use `range(start, stop[, step])` to specify the start, end, and (optionally) step to take.

(The `[, step]` notation in the function signature shows that the `step` argument is optional. It's useful to know such conventions when referring to the docs.)

```
In [4]: range(4,20)
```

```
Out[4]: [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
In [5]: range(4,20,7)
```

```
Out[5]: [4, 11, 18]
```

```
In [6]: range(20,4,-3)
```

```
Out[6]: [20, 17, 14, 11, 8, 5]
```

You can combine `range()` with `len()` to iterate over the indices of a sequence.

```
In [7]: for i in range(len(names)):
        print 'Name', i, 'is', names[i]
```

```
Name 0 is Jamie
Name 1 is Cersei
Name 2 is Jon
Name 3 is Sansa
```

But in such cases, the `enumerate()` function is usually more convenient.

```
In [8]: for i, name in enumerate(names):  
        print 'Name', i, 'is', name
```

```
Name 0 is Jamie  
Name 1 is Cersei  
Name 2 is Jon  
Name 3 is Sansa
```

As you might have guessed, the `enumerate()` function takes a sequence, and returns the (index, value) pairs for each item (the 'pairs' are actually called `tuples`, but more on that later...), and you can assign items from a `tuple` to its own variable in the `for` statement.

```
In [9]: print list(enumerate(names))  
[(0, 'Jamie'), (1, 'Cersei'), (2, 'Jon'), (3, 'Sansa')]
```

Occasionally, you might want to loop over two or more sequences at a time. You can pair the entries with the `zip()` function.

```
In [10]: title = 'Game of Thrones'  
houses = ['Lannister', 'Lannister', 'Snow', 'Stark']  
for char, house, name in zip(title, houses, names):  
    print char, '-', name, house  
  
G - Jamie Lannister  
a - Cersei Lannister  
m - Jon Snow  
e - Sansa Stark
```

Note how `zip()` gracefully fits the iterator to the length of the shortest sequence, i.e., only the first four characters of the string 'Game of Thrones' were iterated.

## break and continue statements

You can manage your loops in more detail using `break` and `continue` statements.

A `break` statement, as the name implies, will break you out of the smallest enclosing loop.

```
In [11]: for name, house in zip(names, houses):  
        if house == 'Snow':  
            break  
        else:  
            print name, house
```

```
Jamie Lannister  
Cersei Lannister
```

A continue statement will simply skip over to the next item in the iterator, instead of breaking out of the loop.

```
In [12]: for name, house in zip(names, houses):  
        if house == 'Snow':  
            continue # compare to the previous example where we stopped  
                the loop at Snow, now we simply skip it  
        else:  
            print name, house
```

```
Jamie Lannister  
Cersei Lannister  
Sansa Stark
```

## Some more data structures

Before we move on, now might be a good time to cover a few more data structures.

### dict (dictionary)

The most useful data structure in python (my very personal opinion)! Also known as *associative arrays* or *hash tables* in other languages, a python dictionary maps *hashable* values to *arbitrary* objects. Dictionaries can be created by placing a comma-separated list of `key:value` pairs within curly braces. Just remember that the key must be immutable (like a string).

```
In [13]: me = {'name':'Jongbin', 'email':'jongbin@stanford.edu'}  
        print me  
  
{'name': 'Jongbin', 'email': 'jongbin@stanford.edu'}
```

You can assign new keys to existing dictionaries.

```
In [14]: me['cel'] = '650-123-4567'  
        print me  
  
{'cel': '650-123-4567', 'name': 'Jongbin', 'email': 'jongbin@stanfor  
d.edu'}
```

Or delete existing `key:value` pairs with the `del` statement.

```
In [15]: del me['email']  
        print me  
  
{'cel': '650-123-4567', 'name': 'Jongbin'}
```

The key of a dictionary can't be a list (because lists are mutable), but the value sure can!

```
In [16]: me['siblings'] = ['Hanbyul', 'Hansol']  
        print me  
  
{'siblings': ['Hanbyul', 'Hansol'], 'cel': '650-123-4567', 'name': 'Jongbin'}
```

Use the `keys()` method of dictionary objects to get a list of the keys used in the dictionary.

```
In [17]: me.keys()  
  
Out[17]: ['siblings', 'cel', 'name']
```

And use the `in` keyword (compatible with all lists) to see if a certain key exists in the dictionary.

```
In [18]: 'name' in me.keys()  
  
Out[18]: True  
  
In [19]: 'email' in me.keys()  
  
Out[19]: False
```

When the keys are simple strings, it is sometimes easier to specify pairs using the `dict` constructor.

```
In [20]: me = dict(name='Jongbin', email='jongbin@stanford.edu', siblings=['Hanbyul', 'Hansol'])  
        print me  
  
{'siblings': ['Hanbyul', 'Hansol'], 'email': 'jongbin@stanford.edu', 'name': 'Jongbin'}
```

The `iteritems()` method lets you loop over each key:value pair.

```
In [21]: for key, value in me.items():  
        print key, ': ', value  
  
siblings : ['Hanbyul', 'Hansol']  
email : jongbin@stanford.edu  
name : Jongbin
```

## tuples

Tuples are pretty similar to lists, except for the fact that they are immutable. They consist of a number of values separated by commas (not necessarily, but often, enclosed in parentheses).

```
In [22]: description = 'male', 'dark hair'
        print description
('male', 'dark hair')
```

```
In [23]: description[0]  # tuples are also sequences, and can be indexed
Out[23]: 'male'
```

```
In [24]: description[1:]  # or sliced
Out[24]: ('dark hair',)
```

```
In [25]: description[0] = 'female'  # but NOT changed, because they are immutable
-----
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-25-193ada17dc41> in <module>()
----> 1 description[0] = 'female'  # but NOT changed, because they are
immutable

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

While being immutable may seem like a minor difference from lists, the implications are quite big, and tuples are generally used for very different purposes compared to lists. For example, tuples can be used as the key for dictionaries (think sparse matrices).

```
In [26]: super_sparse_matrix = {(0, 0):1, (1000, 1000):1}  # a 1000*1000 matrix
        print super_sparse_matrix
{(0, 0): 1, (1000, 1000): 1}
```

```
In [27]: word_matrix = {('apples', 'bananas'):1, ('apples', 'pears'):1}  # a matrix
        print word_matrix
{('apples', 'pears'): 1, ('apples', 'bananas'): 1}
```

There are many more data structures commonly used in python, but lists, dictionaries, and tuples pretty much cover the basics (not to mention that these three constitute enough to fully represent the [JSON \(http://json.org/\)](http://json.org/) format in python)



## List comprehension

List comprehension is python's way of creating lists (and also other data structures) in a concise manner. One way to create a list of squares would be:

```
In [28]: squares = [] # make an empty list
        for x in range(10):
            squares.append(x**2)

        print squares

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

However, the more 'pythonic' way to do this, is to use list comprehension:

```
In [29]: [x**2 for x in range(10)]

Out[29]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The command reads:

```
build a list out of the square of x (x**2), for the values of x in range(10)
```

List comprehension can be used to build a list of tuples too.

```
In [30]: [(x, y) for x in range(10) for y in range(10) if x*y == 21]

Out[30]: [(3, 7), (7, 3)]
```

This is equivalent to the nested for loop:

```
In [31]: twenty_one = []
        for x in range(10):
            for y in range(10):
                if x*y == 21:
                    twenty_one.append((x, y))

        print twenty_one

[(3, 7), (7, 3)]
```

Just be aware that if the item of the list is a tuple, it must be parenthesized.

```
In [32]: [x, y for x in range(10) for y in range(10) if x*y == 21] # this wo
n't work
```

```
File "<ipython-input-32-c0af109a84cc>", line 1
  [x, y for x in range(10) for y in range(10) if x*y == 21] # this
won't work
      ^
SyntaxError: invalid syntax
```

Let's enhance our list of vowels from the previous exercises, by appending the uppercase letters as well.

```
In [33]: vowels = vowels + [V.upper() for V in vowels]
print vowels

['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U']
```

List comprehension can also be used to build dictionaries.

Try building a sparse matrix (represented by a dictionary), where the position is indexed by words  $x$  and  $y$ , and

$$(x, y) = \begin{cases} 1 & \text{if } y \text{ is longer than } x \\ 0 & \text{otherwise} \end{cases}$$

(But note that the example below is not always the best way to do this! Implementation should depend on your context - what do you want to do with the data/matrix?)

```
In [34]: words = ['anti', 'happy', 'evening', 'eagles', 'interior', 'zebra']
{(x,y):1 for x in words for y in words if len(y) > len(x)}
```

```
Out[34]: {('anti', 'eagles'): 1,
('anti', 'evening'): 1,
('anti', 'happy'): 1,
('anti', 'interior'): 1,
('anti', 'zebra'): 1,
('eagles', 'evening'): 1,
('eagles', 'interior'): 1,
('evening', 'interior'): 1,
('happy', 'eagles'): 1,
('happy', 'evening'): 1,
('happy', 'interior'): 1,
('zebra', 'eagles'): 1,
('zebra', 'evening'): 1,
('zebra', 'interior'): 1}
```

## Read/Write Files

Often, you will need to read some data into your python workspace, do something to/with said data, and then write the results to another file. We'll take a look at the most basic file read/write methods, which will get you started with your work, and look at some more advanced topics if we have more time.

## File objects

Think of a python file object as a portal connecting your python workspace to a file on your hard drive. You can open a file object with the built-in `open(filename, mode)` function. The `filename` argument is a string specifying the file name, and the `mode` argument can be one of the following values, specifying whether you want to read from or write to the file:

- `'r'`: read
- `'w'`: write (overwrites any existing files with same filename)
- `'a'`: append (write additional to any existing data)

(you can also open files for both read/write with mode `'r+'`, but this best avoided if possible)

Remember that `open()` simply creates the 'portal', and you have to call additional methods on that file object to either read or write. Since reading can be a little more complicated, let's start with a simple write:

```
In [35]: f = open('example.txt', 'w')
         print f
         <open file 'example.txt', mode 'w' at 0x7ff908384660>
```

Note that after creating the file object, the empty filename file (in the above example, `example.txt`) is created in your working directory. Now, let's actually write something to it:

```
In [36]: f.write('Something')
```

You can only write strings to a file object:

```
In [37]: some_list = [1, 2, 3]
         f.write(some_list)

-----
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-37-70fc2ad98f1e> in <module>()
      1 some_list = [1, 2, 3]
----> 2 f.write(some_list)

TypeError: expected a string or other character buffer object
```

To write anything other than a string, use the `str()` built-in function to convert it to a string first:

```
In [38]: f.write(str(some_list))
```

You might notice that even though you've called `write()` a couple times, the actual file on your hard drive doesn't necessarily get updated. That's because a file object's writes are kept in buffer. To complete all the writes and close the file object, call the `close()` method:

```
In [39]: f.close()
```

Note that using a closed file object will result in an error:

```
In [40]: f.write('...')
```

```
-----
-----
ValueError                                Traceback (most recent call
  last)
<ipython-input-40-8be1c293604e> in <module>()
----> 1 f.write('...')

ValueError: I/O operation on closed file
```

## Reading from a URL

Reading data from a URL in python is pretty simple, using the `urllib2` module. `urllib2` lets you open URLs in read mode, as if they were file objects.

Let's use python's `urllib2` to read Charles Dickens' "A Tale of Two Cities" from <https://goo.gl/fHIeOi> (<https://goo.gl/fHIeOi>)

(This is just for illustration. Note, there are other libraries that are usually more appropriate for reading/scraping web pages.)

```
In [41]: import urllib2 # the import statement is used in python to import mo
          dules/libraries

link = urllib2.urlopen('https://goo.gl/fHIeOi') # open the url
print link
text = link.read()
link.close() # just like file objects, url connections should be clo
sed after you're done

<addinfourl at 140707439963672 whose fp = <socket._fileobject object
at 0x7ff90804d9d0>>
```

The text variable now contains the entire text of "A Tale of Two Cities". For obvious reasons, I'm not going to print the contents of text on the notes, but let's try saving it to a file.

```
In [42]: f = open('two_cities.txt', 'w') # open file object in write mode
         f.write(text)
         f.close()
```

## Reading from file objects

And now, we have a file to practice reading from! We can create a file object just like we did for writing, but with the 'r' mode specified:

```
In [43]: f = open('two_cities.txt', 'r') # open file object in read mode
```

A file object will iterate over the contents of the file it is connected to. For example, the `readline()` method will read the file, one line at a time. And consecutive calls to `readline()` will keep giving you the next line:

```
In [44]: print 'first line:', f.readline() # read the first line
         print 'second line:', f.readline() # read the second line
```

Since the file object essentially provides an iterator over each line of the file, you can loop over the file object line-by-line. This is memory efficient, fast, and leads to simple code:

```
In [45]: n = 1 # a simple counter to control the number of lines printed
for line in f:
    print line
    if n > 10:
        break
    n += 1
```

Just like when writing, don't forget to close files after you're done!

```
In [46]: f.close()
```

As your file I/O gets complex, opening and closing can become quite painful (e.g., what if an error occurs before you close the file object? what happens to the memory it's using?), and forgetting to close file objects is potentially dangerous. So, it's good practice to use the `with` and `as` keywords, which makes sure that the file is properly closed after operations are finished, even if an error occurs during operations:

```
In [47]: with open('two_cities.txt', 'r') as f:
        n = 1
        for line in f:
            print line
            if n > 10: break
            n += 1
```

## Handling exceptions

Sometimes, you will anticipate certain errors, and want your code to behave in a specific manner when such errors occur (e.g., if you're accessing Twitter API, and know that some results will be junk that cause errors). Such errors that occur at runtime (distinguished from syntax errors) are known as exceptions. You can use `try-except` statements to catch certain exceptions.

One common example is assigning a value to an undefined dictionary key. Let's count the occurrence of each alphabet in a certain string:

```
In [48]: s = 'how many wood would a woodchuck chuck if a woodchuck would chuck
         wood'

         d = {} # define an empty dictionary
         for char in s:
             d[char] += 1 # increase the count of d[char] by 1 ... this will
                           result in a KeyError

         print d

-----
-----
KeyError                                Traceback (most recent call
last)
<ipython-input-48-41d1b010857e> in <module>()
      3 d = {} # define an empty dictionary
      4 for char in s:
----> 5     d[char] += 1 # increase the count of d[char] by 1 ... th
is will result in a KeyError
      6
      7 print d

KeyError: 'h'
```

As expected, the above code resulted in a `KeyError` (as we can see from the error message - or `Traceback`). Of course, we could avoid the error by checking if the key exists each time before increasing a count, but this would require an iteration through the entire list of keys every time we increase a value, and that could be potentially expensive if we have many keys. A more efficient way would be to capture the `KeyError` in a `try-except` statement. The `except` statement can specify what type of errors to handle, so we don't end up ignoring other meaningful errors:

```
In [49]: s = 'how many wood would a woodchuck chuck if a woodchuck would chuck
         wood'

         d = {} # define an empty dictionary
         for char in s:
             try: # try everything in this block, and goto the except block i
f an error occurs
                 d[char] += 1 # increase the count of d[char] by 1 ... this w
ill result in a KeyError
             except KeyError: # execute the following block, only if the erro
r is a KeyError, otherwise abort as usual.
                 d[char] = 1 # instead of incrementing, initiate a key:value p
air in the dictionary

         print d

{'a': 3, ' ': 12, 'c': 8, 'd': 6, 'f': 1, 'i': 1, 'h': 5, 'k': 4,
 'm': 1, 'l': 2, 'o': 11, 'n': 1, 'u': 6, 'w': 7, 'y': 1}
```

While catching errors can be useful, always be careful of which errors you catch. Ignoring unexpected error messages can render your results useless!



## Exercise 2.

Hopefully, we've reached this point by the end of the morning session, and now would be a perfect time for an exercise.

1. Declare three lists:

```
names = ['Harry', 'Ron', 'Hermione']
verbs = ['likes', 'hates', 'eats']
objects = ['pie', 'owls', 'the snitch']
```

Use list comprehension to create a list of all (name, verb, object) combinations. (The final result should have length 27.)

2. With the list of combinations from the previous question, use a for loop to print sentences of what Harry eats. (Hint: You can use an and/or statements to check two or more conditions in a single if statement, e.g.,

*if condition\_1 and condition\_2: ...*

The final output should be properly formatted as a sentence, e.g.,

```
...
Harry eats pie.
...
```

3. Create a python dictionary that counts the occurrence of every word, delimited by white spaces, in "A Tale of Two Cities".
4. Find words that occur between 500 and 700 times in "A Tale of Two Cities".