

Object-Oriented Programming (Python)

Data Types, Casting, Operators

Dr. Bianca Schoen-Phelan

bianca.phelan@tudublin.ie

www.biancaphelan.ie



@BSPhelan

KE-3-005a

Objectives

- Discuss differences in programming paradigms
- Discuss the lab: dynamic typing
- Revise variable usage
- Use the different Python data types

Comparison POP and OOP

POP

- Top down approach
- Records
- Procedure
- Module
- Program execution is main concern during design phase
 - Intuitive design
- Breaks a problem or task down into smaller sub-tasks and routines
- Difficult to maintain when changes are implemented

OOP

- Modular approach
- Object
 - Basic unit of OOP
- Method
 - Methods are part of an object
- Class
- Recycling/Reusing of code
- Design strong connection with UML
- Typically allows for more complicated programs while using less code
- Considered better for security applications

From the lab: Python Dynamic Typing

Static (Java, C++)

- Variable doesn't need to be defined before used
- Static has to do with the **explicit** initialisation or declaration of a variable
- Variable needs to be initialized before used

Dynamic (Python)

- Variable must be defined before used
- Does **not** require **explicit** declaration of variable.

Example of Dynamic Typing

Static declaration in C

```
int num, sum;  
// explicit declaration num = 5;  
// now use the variables  
Num = 5;  
sum = 100;  
sum = sum + num;
```

Dynamic declaration in Python

```
Num = 100  
// directly declared and used
```

- Can you think of pros and cons with dynamic typing?

Transition from one Language to Another

- At varying times in your career you will be required to change languages
 - Our industry moves quickly
- Picking up a new language quickly is a valuable skill that we will practice in this course
 - Picking up the syntax is relatively easy
 - You can learn enough to be ‘dangerous’ in one week
 - Proficiency relies on you mastering the aspects of the new language that the previous one didn’t provide
 - Make your code look “Pythonic” rather than C written in Python!

Comparing Python to Python

Pythonish

```
i = 0
while i < len(myItems):
    print(myItems.get(i)) i += 1
```

Pythonic

```
for item in my_items:
    print(item)
```

Revision on variables

- Names given to data
- Store information
- Also for manipulation of information
- Examples:
 - Age of a person: `user_age = 30`
 - Name of a person: `user_name = 'Bianca'`
 - Or `user_age, user_name = 30, 'Bianca'`

Naming Variables

- Allowed are only certain letters
 - a-z and A-Z
- Numbers
- Underscores `_`
- First character cannot be a number
 - `user_name`
 - `user_name2`
 - **NOT** `2user_Name`
- Cannot use reserved names, for example print, if, else, while
- Case sensitive `userName` and `UserName` are not the same

Two conventions:

- Camel Case:

`myName`

- Underscore:

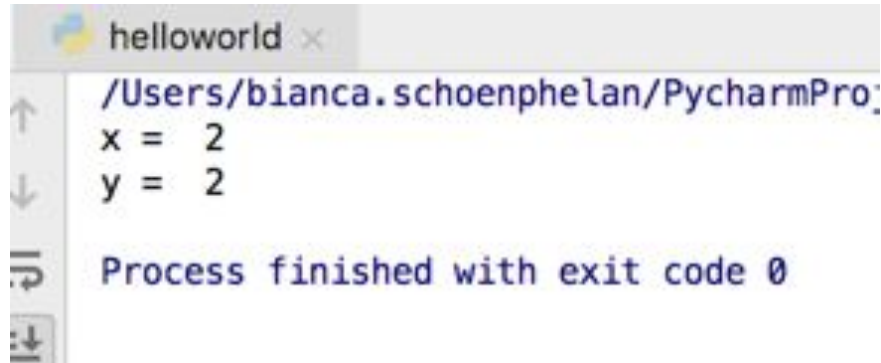
`my_name`

Underscore is recommended in Python!

Assigning Values to Variables

- Using the assignment operator =

```
x = 3
y = 2
x = y
print ("x = ", x)
print ("y=" , y)
```

A terminal window titled 'helloworld' showing the execution of the Python code. The output is: /Users/bianca.schoenphelan/PycharmPro: x = 2 y = 2. Below the code, it says 'Process finished with exit code 0'.

```
helloworld x
/Users/bianca.schoenphelan/PycharmPro:
x = 2
y = 2

Process finished with exit code 0
```

Basic Operators

Symbol	Operation	Example x=5, y=2	Result
+	Addition	x + y	7
-	Subtraction	x - y	3
*	Multiplication	x * y	10
/	Division	x / y	2.5
//	Floor division	x // y	2
%	Modulo	x % y	1
**	Exponent	x ** y	25

- More operators:

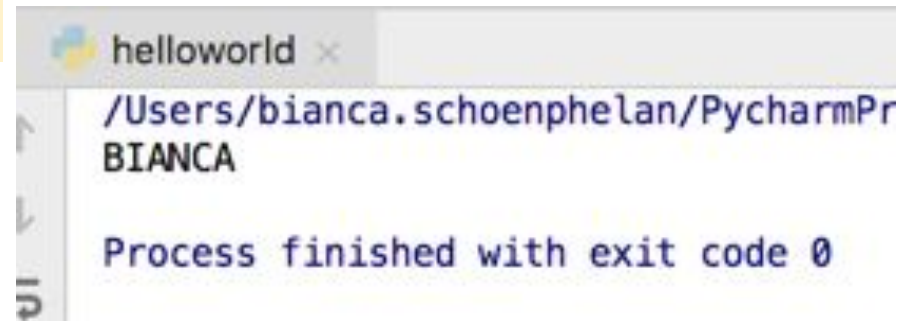
- +=
- -=
- *=

Python doesn't
do ++ or --

Data Types

Data Types

- Integer, example: `user_age = 30`
- Float, example: `user_height = 1.55`
- String, example: `user_name = 'Bianca', user_age = '30'`
 - Many built in functions for String operations, example:
`print('Bianca'.UPPER)`



A terminal window titled 'helloworld' showing the execution of the Python code. The path '/Users/bianca.schoenphelan/PycharmPr' is visible. The output is 'BIANCA' in all caps. Below the output, it says 'Process finished with exit code 0'.

Data Types: String

- String

```
variable_name = 'initial Value' or
```

```
variable_name = "initial Value"
```

- We can concatenate substrings with +

- Example:

```
my_name = 'Bianca'
```

```
conjunction = 'and'
```

```
his_name = 'Don'
```

```
print(my_name+conjunction+his_name)
```

Strings are considered immutable. What does that mean?

Formatting Strings with %

- We can change the look of a String by manipulating the output with the % operator
- “string to be formatted” % (values or variable to be inserted to string, separated by comma)
 - There are 3 parts to this operation
 - Round bracket part to the operation is known as a tuple

- Example:

Placeholder

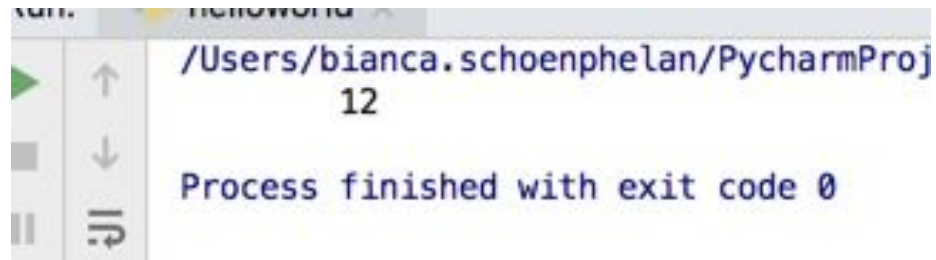
```
message = ("The price of this %s smartphone is %d EUR and  
the exchange rate to British Pound is %4.2f to 1 EUR" %  
(brand, 555, exchange_rate))
```



```
in: helloworld x  
/Users/bianca.schoenphelan/PycharmProjects/TestProject1/venv/bin/python /Users/bianca.schoenphelan/PycharmF  
The price of this Samsung smartphone is 555 EUR and the exchange rate to British Pound is 1.12 to 1 EUR  
Process finished with exit code 0
```

Placeholders and Formatters

- Placeholders will be replaced with values
- %s formatter represents a string
- %d formatter represents an integer
 - We can include spaces in the representation with giving a number of spaces
 - Example: `print("%10d"%(12))`



```
run. /Users/bianca.schoenphelan/PycharmProj
12
Process finished with exit code 0
```


Formatters

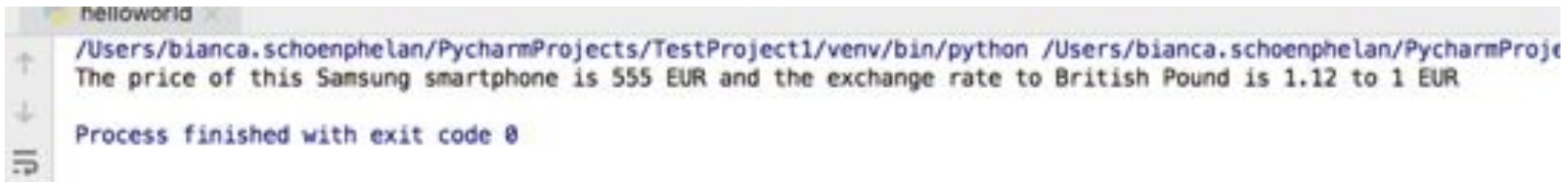
- %f formats decimal numbers, example uses 4.2f:
 - 4 = total length
 - 2 = decimal positions
 - What does the following do:

```
print ("%7.2f"% (1.123) )
```

format() method

- “string to be formatted”.format(values or variables to be inserted into string, separated by comma)
- No more %s %d %f for placeholders, but now {}
- Example:

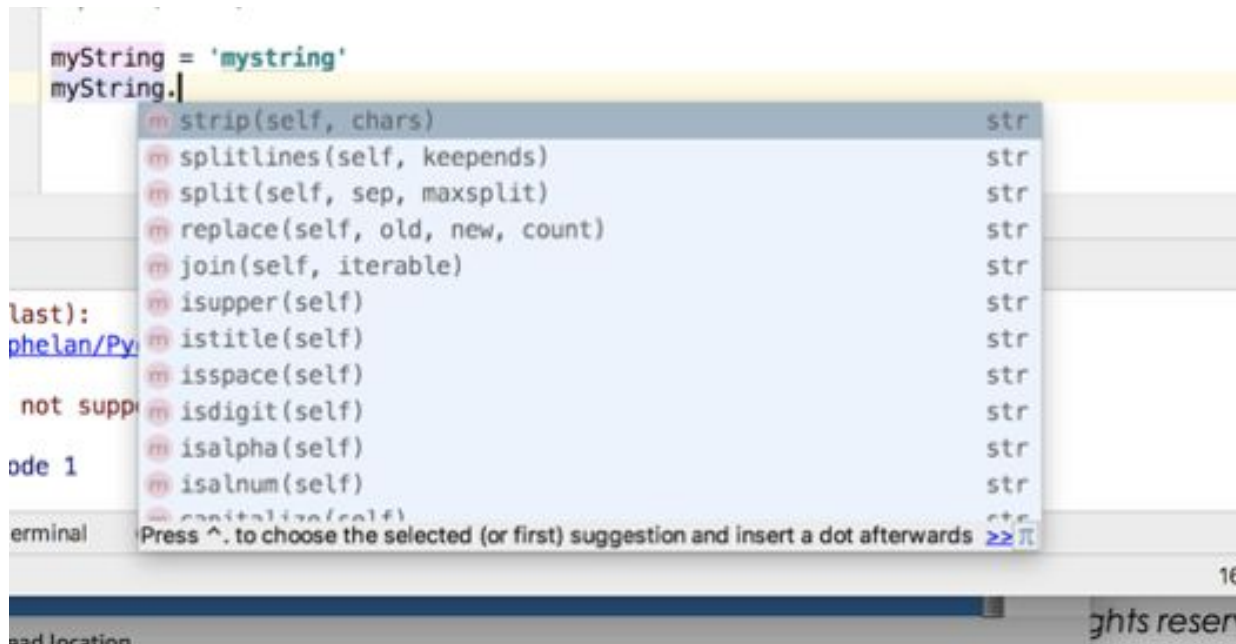
```
$ message = "The price of this {0:s} smartphone is  
{1:d} EUR and the exchange rate to British Pound is  
{2:4.2f} to 1 EUR".format('Samsung', 555,  
exchange_rate)  
  
$ print(message)
```



```
helloworld  
/Users/bianca.schoenphelan/PycharmProjects/TestProject1/venv/bin/python /Users/bianca.schoenphelan/PycharmProje  
The price of this Samsung smartphone is 555 EUR and the exchange rate to British Pound is 1.12 to 1 EUR  
  
Process finished with exit code 0
```

String functions

There are many string functions available



Some examples: count

- `count(sub, [start, [end]])`
 - Start and end are optional parameters
 - Return the number of times the substring sub appears in a string
 - Function is case sensitive
- What does the following do:

```
1. print('this is a string'.count('s'))
```

```
2. print('this is a string'.count('s',4))
```

```
3. print('this is a string'.count('s',4, 10))
```

```
4. print('This is a string'.count('T'))
```

More examples: endswith

- `endswith(suffix, [start, [end]])`
 - Returns true if the string ends with a specified suffix, otherwise returns false
 - Suffix can also be a tuple of suffixes
 - Function is case sensitive
- What does the following do:

```
1. print('Postman'.endswith('man'))
```

```
2. print('Postman'.endswith('man', 3))
```

```
3. print('Postman'.endswith('man', 2, 6))
```

```
4. print('Postman'.endswith('man', 2, 7))
```

```
5. print('Postman'.endswith('man', 'ma', 2, 6))
```

String examples: find/index

- `find/index(sub, [start, [end]])`
 - Returns the index in a string, where the **first occurrence** of the substring is found
 - `find()` returns -1 if sub not found
 - `index()` returns `ValueError` if sub not found
- What does the following do:
 1. `print('This is a string'.find('s'))`
 2. `print('This is a string'.find('s',4))`
 3. `print('This is a string'.find('s',7,11))`
 4. `print('This is a string'.find('o'))`
 5. `print('This is a string'.index('o'))`

More string functions

Alphanumeric is either a letter or a number.

- `isalnum()`
 - Returns true if all characters in a string are alphanumeric, and there is at least one character (false or otherwise)
 - Does not include whitespaces
- What does the following do:
 1. `print('abc123'.isalnum())`
 2. `print('a b c 1 2 3'.isalnum())`
 3. `print('abc'.isalnum())`
 4. `print('123'.isalnum())`

More string functions cont'd

- `isalpha()`
 - Returns true if all characters in the string are alphabetic and there is at least one character (true or otherwise)
- What does the following do:
 1. `print('abc'.isalpha())`
 2. `print('abc123'.isalpha())`
 3. `print('123'.isalpha())`
 4. `print('a b c'.isalpha())`

More string functions cont'd

- `isdigit()`
 - Returns true if all characters in the string are digits, and there is at least one character (true or otherwise)
- What does the following do:
 1. `print('123'.isdigit())`
 2. `print('123abc'.isdigit())`
 3. `print('abc'.isdigit())`
 4. `print('1 2 3'.isdigit())`

isnumeric is TRUE if we feed it a digit from any language Python recognises.

```
>>> print('二'.isdigit())
False
>>> print('二'.isnumeric())
True
```

All decimals are digits, but not all digits are decimals.

Similar functions

- `islower()`
- `isspace()`
- `istitle()`
- `isupper()`

Working with many strings

- `join()`
 - Returns a string where the parameter provided is joined by a separator
- What does the following do:

```
seperator = '-'
```

```
my_tuple = ('a', 'b', 'c')
```

```
my_list = ['d', 'e', 'f']
```

```
my_string = "Hello World"
```

```
print(seperator.join(my_tuple))
```

```
print(seperator.join(my_list))
```

```
print(seperator.join(my_string))
```

Working with many strings cont'd

- `replace(old,new[,count])`

- Returns a copy of a string with all occurrences of substring old replaced by substring new
- Count is options, means that only these amount of instances are replaced
- The function is case sensitive

- What does the following do:

```
1.print('This is a string'.replace('s','p'))
```

```
2.print('This is a string'.replace('s','p',2))
```

Working with strings

- `split([sep, [maxsplit]])`
 - Returns a list of words in a string
 - If `sep` is not given, use whitespace
 - If `maxsplit` is given, this is the maximum number of splits performed
 - Function is case sensitive
- What does the following do:
 1. `print('This, is, a, string'.split(','))`
 2. `print('This is a string'.split())`
 3. `print('This, is, a, string'.split(', ', 2))`

More strings

- `splitlines([keepends])`

- Returns a list of lines, break at boundary
- Linebreaks are not included unless `keepends` is used and `true`

- What does the following do:

1. `print("This is the first line. \nThis is the second line.".splitlines())`
2. `print("This is the first line."
"This is the second line.".splitlines())`
3. `print("This is the first line. \nThis is the second line.".splitlines(True))`

Non printing characters

- Can be inserted directly into a string
 - New line: `\n`
 - Tab: `\t`
- Apostrophes in Names need to be escaped
 - Example: `O\'Brien`

Last of the string functions for today

- `strip([chars])`
 - Returns a copy of a string with the leading and trailing characters char removed
 - If char is not provided, the function uses whitespaces
- What does the following do:
 1. `print(" This is a string ".strip())`
 2. `print("This is a string".strip('s'))`
 3. `print("This is a string".strip('g'))`

Data Types: List

- Zero-indexed
- Guarantees order of items

List: collection of data which are normally related, can be different data types

```
list_name = [initial values]
```

```
user_age = [10, 20, 30, 40, 50, 60, 70], or
```

```
user_age = []
```

This is an empty list. Add items using append() method

- Individual items can be accessed by an index that starts with 0 (never with 1)
 - Last item in the list has the index -1, second last -2, etc
 - user_age[-1] will return 70
- Can be assigned to a variable:

```
user_age2 = user_age
```

```
user_age3 = user_age[1:3]
```

```
user_age4 = user_age[1:6:2]
```

So called **slice** notation:

Item at the start is always **included**, item at the end is always **excluded**. Third number is a stepper (every 2nd item) in range.

Slices

```
user_age = [10, 20, 30, 40, 50, 60, 70]
```

- Useful defaults:
 - First number is 0, example: `user_age [:4]` returns values from 0 to 4-1
 - `user_age[1:]` returns values from index 1 to length-1

Working with Lists

- Modify by assigning new value to an index

```
user_age[3] = 99 results in
```

```
user_age = [10, 20, 30, 99, 50, 60, 70]
```

- Add an item with append()
- `user_age.append(100)` results in

```
user_age = [10, 20, 30, 99, 50, 60, 70, 100]
```

- Remove an item with del

- `Del user_age[3]` results in

```
user_age = [10, 20, 30, 50, 60, 70, 100]
```

Working with Lists

```
my_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',  
'i', 'j']
```

```
1.del my_list[2]
```

```
print(my_list)
```

```
2.del my_list[1:5]
```

```
print(my_list)
```

```
3.del my_list[:3]
```

```
print(my_list)
```

```
4.del my_list[2:]
```

```
print(my_list)
```

Extending lists

- `extend()`
 - Combine two lists
- What does the following do:

```
my_list1 = ['a', 'b', 'c']
```

```
my_list2 = [1, 2, 3]
```

```
my_list1.extend(my_list2)
```

```
print(my_list1)
```

Working with lists

- `in`



No brackets

- True if the parameter is in the list, false otherwise
- `print('c' in my_list1)` returns true

- `insert()`

- Add an item at a particular position

- `len()`

- Number of items

Working with Lists cont'd

- `pop()`
 - Get an item at an index position and remove it from the list
- Example:

```
my_list1 = ['a', 'b', 'c']  
member = my_list1.pop(2)  
print(member)  
print(my_list1)
```

Removes the last item on the list if no argument is given.

Working with lists

- `remove()`
 - Removes an item from a list and requires a specific value
- Example:

```
my_list1 = ['a', 'b', 'c']  
my_list1.remove('a')  
print(my_list1)
```

Removes the last item on the list if no argument is given.

So what's the difference?

- What is the difference between del, pop and remove?
 - **remove**: removes first matching value from the List (not a specific index location)
 - **del**: removes an item at a specific index location
 - **pop**: Removes an item at a specific index location and returns it

Sorting lists

- `reverse`
- `sort`
- `sorted`
 - Returns a new sorted list without changing the original list

Operators and Lists

- + operator to concatenate

```
my_list1 = ['a', 'b', 'c']
```

```
my_list2 = [1, 2, 3]
```

```
print(my_list1+my_list2)
```

- * operator to multiply

```
print(my_list1*3)
```

Data Types: Tuples

- **Tuples:** like lists, but values cannot be changed

```
tuple_name = (values)
```

```
months_of_the_year = ('Jan', 'Feb', 'Mar')
```

- Individual values can be accessed through indexes, like we did with lists
- Delete a complete tuple with del

```
del months_of_the_year
```

```
print (months_of_the_year)
```

Will produce an error if the variable name is not defined!

- Indexes can be used the same way as with lists

Working with Tuples

- `del`: deletes the whole tuple
- `in`: “a” in myTuple -> True
- `len()`
- `+` concatenates tuples
- `*` multiplies tuples

The tuple itself does not get modified.

Data Types: Sets

- Similar and hugely different
- Cannot contain duplicates
- Can contain mixture of types
- Are unordered!
- Example:

```
hello = set('hello')
```

```
print(hello)
```

```
-> {'l', 'h', 'e', 'o'}
```

Data Types: Dictionary

- **Dictionary:** collection of related data PAIRS

```
dictionary_name = {dictionaryKey : data}
```

```
customers = {"Bianca":30, "Brian":31, "Susan":32}
```

- Dictionary key must be unique, error when you try

```
customers = {"Bianca":30, "Brian":31, "Bianca":21}
```

- Different python versions will behave differently here and ask you to use a constructor instead

- Dictionaries are collections, but they are not sequences
 - There is no order
 - The order might change if the elements in the dictionary change

Working with Dictionaries

```
print(users_and_age["Bianca"])
```

30

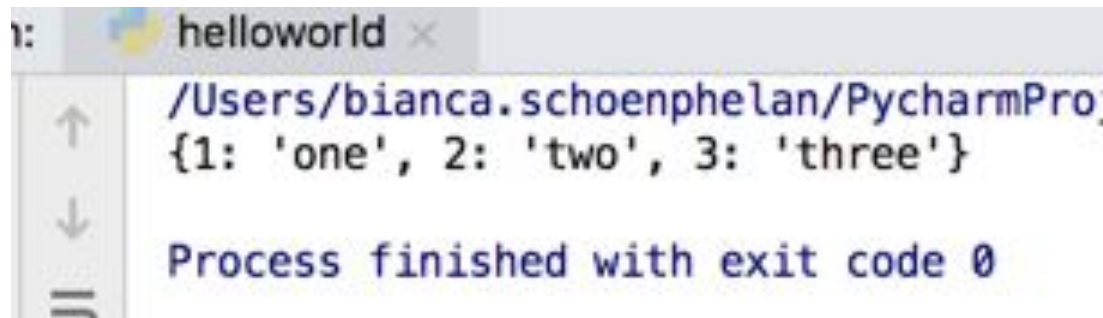
- Modify: `users_and_age["Bianca"] = 51`
- Remove: `del users_and_age["Bianca"]`
- Empty: `users_and_age.clear()`
- `get()`: returns a value for a given key
- `in`: check if an item is in the dictionary
- `items()`: returns a list of dictionary pairs as tuples
- `print(users_and_age.items())`

```
dict_items([('Bianca', 30), ('Brian', 40),  
('Susan', 50), ('Lana', 'Not Available')])
```


Working with Dictionaries

- `keys()`: returns a list of dictionary keys
- `values()`: returns a list of dictionary values
- `len()`: number of items in dictionary
- `update()`
- Example:

```
dict1 = {1: 'one', 2: 'two'}  
dict2 = {1: 'one', 3: 'three'}  
dict1.update(dict2)  
print(dict1)
```



```
helloworld x  
/Users/bianca.schoenphelan/PycharmPro  
{1: 'one', 2: 'two', 3: 'three'}  
  
Process finished with exit code 0
```

Type Casting

- When we convert a variable's data type from one data type to another we talk about **type casting**
- Python offers 3 built in functions for type casting:
 - `int()` : takes a float or appropriate string and converts it into an integer
 - `float()` : takes an integer or an appropriate string and converts it into a float
 - `str()` : converts an integer or a float into a string

Summary

- ★ Basic Data Types
- ★ Advanced Data Types
- ★ Working with the types
- ★ Casting



References

1. [Kenny Eliason, Difference between OOP and Procedural Programming, https://neonbrand.com/website-design/procedural-programming-vs-object-oriented-programming-a-review/](https://neonbrand.com/website-design/procedural-programming-vs-object-oriented-programming-a-review/), 2013, Accessed Sep 2018.
2. [The Real Python, 2012-2018, https://realpython.com/switching-to-python/](https://realpython.com/switching-to-python/), Accessed Sep 2018.
3. Learn Python in one day, Jamie Chan, 2014
4. Moutaz Haddara, Introduction to Object-oriented programming, slideshare, 2014.
5. Jamie Chan, Learn Python in one day, 2014.