ELEG4701

Intelligent Interactive Robot Practice

Basic Elements in Python

Jiewen Lai
Research Assistant Professor
EE, CUHK
jiewen.lai@cuhk.edu.hk

Please be noted…

- You need some Python / C++ knowledge for ROS
- We will only have this **ONE** lecture for **BASIC** Python
- You need to practice more by yourself

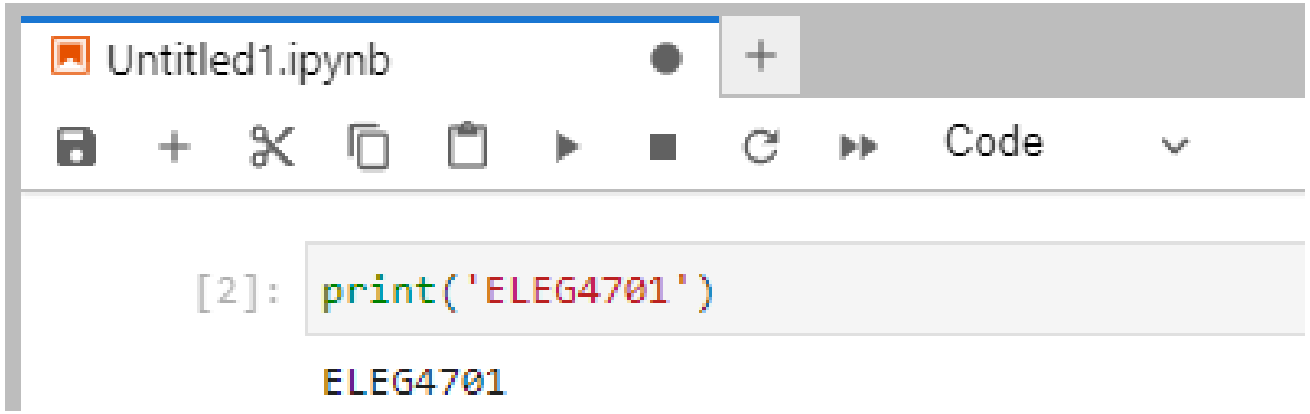
- While I am talking, please try the **practices** at the same time (highlighted in the ORANGE color)

… We will have a lab sheet for today's course

Installation-free py environment

Jupyter Notebook:

https://jupyter.org/try-jupyter/lab/?path=notebooks%2FIntro.ipynb



Variables

- Variables in Python can contain alpha-numeric characters and some special characters.
- By convention, it is common to have variable names that start with **lowercase letters** and have class names beginning with a **capital letter**; but you can do whatever you want.
- Some keywords are reserved and cannot be used as variable names due to them serving an in-built Python function; i.e., and, continue, break. Your IDE will let you know if you try to use one of these.
- Python is dynamically typed; the type of the variable is derived from the value it is assigned.

Variable Types

- Integer (int)
- Float (float)
- String (str)
- Boolean (bool)
- Complex (complex)
- […]
- User defined (classes)

- A variable is assigned using the "=" operator, i.e.,

In:

```
intVar = 5
floatVar = 3.2
stringVar = 'Food'

print(intVar)
print(floatVar)
print(stringVar)
```

Out:

```
5
3.2
Food
```

- The print() function is used to print something to the screen.

Try them on Jupyter

- Create an integer, float, and string variable.
- Print these to the screen.
- Play around using different variable names, etc.

- You can always check the type of a variable using type() function, e.g.,

In:

```
var = 100
print(type(var))
```

Out:

```
<class 'int'>
```

- Check the type of one of your variables.

- Variables can be cast to a different type

In:

```
share_of_rent = 295.50/2.0
print('1:', share_of_rent)
print(type(share_of_rent))
rounded_share = int(share_of_rent)
print('2:', rounded_share)
print(type(rounded_share))
```

Out:

```
1: 147.75
<class 'float'>
2: 147
<class 'int'>
```

Arithmetic operators

The arithmetic operators:

- Addition: +
- Subtract: -
- Multiplication: *
- Division: /
- Power: **

- Write a couple of operations using the arithmetic operators, and print the results to the screen.

In:

```
print(5+5)

x = 2
y = 10
print(x/y)
```

Out:

```
10
0.2
```

A quick note on
increment operator shorthand

- Python has a common idiom - that is not necessary - but which is used frequently and is worth noting:

```
x += 1
```

is the same as:

```
x = x + 1
```

- This also works for other operators:

```
x += y        # adds y to the value of x
x *= y        # multiples x by the value y
x -= y        # subtracts y from x
x /= y        # divides x by y
```

Boolean operators

Boolean operators are useful when making conditional statements:

- and
- or
- not

Comparison operators

- Greater than: >
- Less than: <
- Greater than or equal to: >=
- Less than or equal to: <=
- is equal to: ==

- Write a couple of operations using comparison operators:

In:

```
intVar = 5
floatVar = 3.2
stringVar = "Food"

if intVar > floatVar:
    print("Yes")

if intVar == 5:
    print("A match!")
```

Out:

```
Yes
A match!
```

Working with strings

In:
```
greeting = 'Hello, Lew!'
print('1:', greeting)
print('2:', len(greeting))
print('3:', greeting[0])
print('4:', greeting[-1])
greeting = greeting.replace("Lew", "class")
print('5:', greeting)
```

Out:
```
1: Hello, Lew!
2: 11
3: H
4: !
5: Hello, class!
```

```
string1 = "Hello"
string2 = "world"
print("1:", string1, string2)
cost = float(35.28)
print("Bar tab = £%f" %cost)
```

```
1: Hello world
Bar tab = £35.280000
```

- Create a string variable
- Work out the **string length**

Dictionaries

- Dictionaries are lists of key-valued pairs

In:

```
prices = {"Eggs": 2.30,
          "Steak": 13.50,
          "Bacon": 2.30,
          "Beer": 14.95}
print("1:", prices)
print("2:", type(prices))
print("The price of bacon is:", prices["Bacon"])
```

Using curly bracket { , , , } for dict.

Indicate the value by name

Out:

```
1: {'Eggs': 2.3, 'Steak': 13.5, 'Bacon': 2.3, 'Beer':
14.95}
2: <class 'dict'>
The price of bacon is: 2.3
```

Indexing

- Indexing in Python is **0-based**, meaning that the first element in a string, list, array, etc., has an index of 0. The second element then has an index of 1, and so on.

In:
```
test_string = "Dogs are better than cats"
print('First element:', test_string[0])
print('Second element:', test_string[1])
```

Out:
```
First element: D
Second element: o
```

You can cycle backward through a list, string, array, etc., by placing **a minus symbol** in front of the index location.

In:
```
test_string = "Dogs are better than cats"
print('Last element:', test_string[-1])
print('Second to last element:', test_string[-2])
```

There is no “-0”

Out:
```
Last element: s
Second to last element: t
```

Indexing

In:

```
test_str = 'Dogs are better than cats'
print('Second to last element:', test_str[4:])
print('The zero-th to forth element:', test_str[:4])
```

Out:

```
Second to last element: are better than cats
The zero-th to forth element: Dogs
```

In:

```
test_str = 'Dogs are better than cats'
print(test_str[5:10])
```

Out:

```
are b
```

1. Create a string that is 10 char in length
2. Print the **second char** to the screen
3. Print the third to last char to the screen
4. Print all char after the fourth char
5. Print char 2-8

Tuples

- Tuples are containers that are immutable; i.e., their contents **CANNOT be altered** once created.

In:

```
tuple1 = (5, 10)
print('1:', tuple1)
print('2:', type(tuple1))
```

Using round brackets for tuples

Out:

```
1: (5, 10)
2: <class 'tuple'>
```

In:

```
tuple[1] = 6
```

Cannot be changed

Out:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[16], line 1
----> 1 tuple[1] = 6

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

Lists

- Lists are essentially constrainers of arbitrary type.
- They are probably the container that you will use most frequently.
- The elements of a list can be of different types
- The difference between **tuples** and **lists** is in **performance**; it is much **faster** to 'grab' an element stored in **tuple**, but lists are much more versatile
- Note that lists are denoted by [ ], and not the () used by tuples

Using squared brackets [] for list

In:

```
numbers = [1, 2, 3]
print("List 1:", numbers)
print("Type of list 1:", type(numbers))
arbitrary_list = [1, numbers, "Hello"]
print("Arbitrary list:", arbitrary_list)
print("Type of arbitrary list:", type(arbitrary_list))
```

Out:

```
List 1: [1, 2, 3]
Type of list 1: <class 'list'>
Arbitrary list: [1, [1, 2, 3], 'Hello']
Type of arbitrary list: <class 'list'>
```

- Create a list and populate it with some different elements.

Adding elements to a list

- Lists are mutable; i.e., their contents **can be changed**. This can be done in many ways
- For example, by using **<u>an index</u>** to replace a current element with a new one

In:

```
numbers = [1, 2, 3]
print("List 1:", numbers)
numbers[1] = 5
print("Amended list 1:", numbers)
```

Out:

```
List 1: [1, 2, 3]
Amended list 1: [1, 5, 3]
```

- You can use the insert() function to add an element to a list at a specific indexed location, **without overwriting** any of the original elements.

In:

```
numbers = [1, 2, 3]
print("List 1:", numbers)
numbers.insert(2, 'Surprise!')
print("Amended list 1:", numbers)
```

Out:

```
List 1: [1, 2, 3]
Amended list 1: [1, 2, 'Surprise!', 3]
```

- Use insert() to put the integer 3 after the 2 that you just added to your string.

- You can add an element to the end of a list using the append() function.

In:

```
numbers = [1, 2, 3]
print("List 1:", numbers)
numbers.append(4)
print("Amended list 1:", numbers)
```

Out:

```
List 1: [1, 2, 3]
Amended list 1: [1, 2, 3, 4]
```

Use append() to add the string “end” as the last element in your list.

Removing elements from a list

- You can remove an element from a list based on the **element value** by using remove()
- Reminder: if there is more than 1 element with this value, only **the first occurrence** will be removed.

In:

```
numbers = [1, 2, 3, 3]
print("List 1:", numbers)
numbers.remove(3)
print("Amended list 1:", numbers)
```

Out:

```
List 1: [1, 2, 3, 3]
Amended list 1: [1, 2, 3]
```

- It is a better practice to remove elements by their **index** using the del() function.

In:

```
numbers = [1, 2, 3, 4]
print("List 1:", numbers)
del numbers[1]
print("Amended list 1:", numbers)
del numbers[-1]
print("Amended list 2:", numbers)
```

Out:

```
List 1: [1, 2, 3, 4]
Amended list 1: [1, 3, 4]
Amended list 2: [1, 3]
```

Use del() to remove the 3 that you added to the list earlier.

For Loops

- The **for loop** is used to iterate over elements in a sequence, and is often used when you have a piece of code that you want to repeat a number of times.
- For loops essentially say:

"For all elements in a sequence, do something"

Example: for loop

- We have a list of species:

```
species = ['dog', 'cat', 'shark', 'falcon', 'deer', 'tyrannosaurus rex']
for i in species:
    print(i)
```

- The for-loop then cycles through each entry in the list, and prints the animal's name to the screen.

```
dog
cat
shark
falcon
deer
tyrannosaurus rex
```

- Note: The i is quite arbitrary. You could just as easily replace it with 'animal', 't', or anything else.

Another Example

- We can also use for-loops for operations, other than printing. For example:

```
numbers = [1, 20, 18, 5, 15, 160]
total = 0
for value in numbers:
    total = total + value
print(total)
```

```
219
```

- Using the list you made a moment ago, use a for-loop to print each element of the list to the screen in turn.

The range() function

- The range() function generates a list of numbers, which is often used to iterate over within for loops.
- The range() function has two sets of parameters to follow:

`range(stop)`

stop: number of int to generate, starting from 0

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Also, pay attention to the indent

`range(start, stop, step)`

start: starting number of the sequence

stop: generate numbers up to, but not including this number

step: the difference between each number in the sequence

```
for i in range(3,6):
    print(i)
```

```
3
4
5
```

```
for i in range(4, 10, 2):
    print(i)
```

```
4
6
8
```

Note:

- All parameters must be **integers**.
- Parameters can be positive or negative.
- The range() function (and Python in general) is 0-index based, meaning list indexes start at 0, not 1. E.g., The syntax to access the first element of a list is mylist[0]. Therefore, the last integer generated by range() is up to, but not including, stop.

- Create an empty list

```
new_list = []
```

- Use the range() and append() functions to add the int 1-20 to the empty list.

```
for i in range(1, 21):
    new_list.append(i)
```

- Print the list, what do you have?

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

break() function

- To terminate a loop, you can use the break() function
- The break() statement breaks out of **<u>the innermost</u>** enclosing for or while loop

```
for i in range(1, 10):
    if i == 3:
        break
    print(i)
```

```
1
2
```

continue() function

- The continue() statement is used to tell Python to **skip** the rest of the statements **in the current loop block**, and then to continue to the next iteration of the loop

```
for i in range(1, 10):
    if i == 3:
        continue
    print(i)
```

```
1
2
4
5
6
7
8
9
```

While loops

- The **while loop** tells the computer to do something – as long as **a specific condition** is met.
- The while loops essentially say:

"While this is true, do this"

- When working with while loops, it's important to remember the nature of various operators.
- A while-loop uses the break() and continue() functions in the same way as a for-loop does.

An example

```
species = ['dog', 'cat', 'shark', 'falcon', 'deer', 'tyrannosaurus rex']
i = 0
while i < 3:
    print(species[i])
    i = i + 1
```

```
dog
cat
shark
```

0 = dog
1 = cat
2 = shark
3 is not happening

A bad example

```
counter = 0
while counter <= 100:
    print(counter)
    counter + 99
```

Inf loop: because counter is always 0

For loop vs. while loop

- You will use for loops **more often** than while loops.
- The for loop is the natural choice for cycling through a list, characters in a string, etc.; basically, **anything of *determinate* size**. (you know the size)

- The while loop is the natural choice if you are cycling through something, such as a sequence of numbers, an ***indeterminate* number** of times until some specific condition is met. (you may not know the size)

Nested loops

- In some situations, you may want a loop within a loop.
- This is known as a nested loop.

- What will the code on the right produce?
- Recreate this code and run it, what do you get?

```
for x in range(1,11):
    for y in range(1,11):
        print(x,'*' ,y, '=',x*y)
```

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
1 * 10 = 10
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
```

...

```
9 * 10 = 90
10 * 1 = 10
10 * 2 = 20
10 * 3 = 30
10 * 4 = 40
10 * 5 = 50
10 * 6 = 60
10 * 7 = 70
10 * 8 = 80
10 * 9 = 90
10 * 10 = 100
```

Conditionals

- There are three main conditional statements in Python:
  - if
  - else
  - elif (meaning else-if)
- We have already used if when looking at the while loops

```
school_night = True
if school_night == True:
    print("No beer")
else:
    print("You may have beer")
```

```
No beer
```

```
school_night = False
if school_night == True:
    print("No beer")
else:
    print("You may have beer")
```

```
You may have beer
```

An example of elif

```
Lew_is_tired = False
Lew_is_hungry = True
if Lew_is_tired is True:
    print("Lew has to teach")
elif Lew_is_hungry is True:
    print("No food for Lew")
else:
    print("Go on, have a biscuit")
```

```
No food for Lew
```

Functions

- A function is **a block of code** that only runs when it is called.
- They are useful if you have operations that need to be done repeatedly; e.g., calculations.
- The function must be **defined <u>before</u>** it is called. In other words, <u>the block of code that makes up the function</u> **must come before** <u>the (main) block of code</u> that makes use of the function.

```
def practice_function(a, b):
    answer = a * b
    return answer

x = 5
y = 4
calculated = practice_function(x, y)
print(calculated)
```

```
20
```

- Create a function that **takes two inputs**, **multiplies them**, and then **returns the result**.
- It should look be like:

```
def fun_name(a, b):
    do something
    return something
```

```
def multiply_function(a, b):
    result = a * b
    return result
```

Try this yourself…

- Create **two different lists of integers**.
- Using your function, write a **nested for-loop** that cycles through each entry in the first list, and multiples it by each of the entries in the second list, and prints the result to the screen.

```
def multiply_function(a, b):
    result = a * b
    return result

number_list = [1, 2, 3]
multiplier_list = [2, 4]
for n in number_list:
    print('_____')
    for m in multiplier_list:
        curr_ans = multiply_function(n ,m)
        print('The answer to ', n, '*', m, 'is: ', curr_ans)
```

```
_____
The answer to  1 * 2 is:  2
The answer to  1 * 4 is:  4

_____
The answer to  2 * 2 is:  4
The answer to  2 * 4 is:  8

_____
The answer to  3 * 2 is:  6
The answer to  3 * 4 is:  12
```

Multiple returns

- You can have a function return multiple outputs in order.

```
def multiply_function(a, b):
    result = a * b
    result2 = result **2
    return result, result2

number_list = [1, 2, 3]
multiplier_list = [2, 4]
for n in number_list:
    print('_____')
    for m in multiplier_list:
        curr_ans, curr_ans2 = multiply_function(n ,m)
        print('The answer to ', n, '*', m, 'is:', curr_ans)
        print('The result of this squared is:', curr_ans2)
```

```
_____
The answer to  1 * 2 is: 2
The result of this squared is: 4
The answer to  1 * 4 is: 4
The result of this squared is: 16
_____
The answer to  2 * 2 is: 4
The result of this squared is: 16
The answer to  2 * 4 is: 8
The result of this squared is: 64
_____
The answer to  3 * 2 is: 6
The result of this squared is: 36
The answer to  3 * 4 is: 12
The result of this squared is: 144
```

Reading and writing to files in Python: The file object

- File handling in Python can easily be done with the **built-in object file**.
- The file object provides all basic functions necessary in order to manipulate files.

- Open up notepad or notepad++.
- Write some text and save the file to a location and with a name you'll remember.

open() function

- Before you can work with a file, you first have to open it using Python's built-in open() function.
- The open() function takes two arguments: (1) the **name** of the file that you wish to use, and (2) the **mode** for which we would like to open the file

```
practiceFile = open('Practice_file_for_IOC.txt', 'r')
```

- By default, the open() function opens a file in 'read mode'
- There are a number of different file opening modes. The most common are: 'r' = read, 'w' = write, 'r+' = both reading and writing, and 'a' = appending.
- Use the open() function to read the file in.

close() function

- Likewise, once you're done working with a file, you can close it with the close() function.
- Using this function will free up any system resources that are being used up by having the file open.

```
practiceFile.close()
```

Reading a file and printing to screen

- Using what you have now learned about for loops, it is possible to **open a file for reading** and then **print each line in the file** to the screen using a for loop.

```
practiceFile = open('practice_file.txt', 'r')
for line in practiceFile:
    print(line)
```

```
The first line of text

The second line of text

The third line of text

The fourth line of text

I'm bored now, you get the idea
```

read() function

- However, you don't need to use any loops to access file contents.
- Python has three in-built file reading commands:

1. **<file>.read()** = Returns the entire contents of the file as a single string:

```
practiceFile = open('practice_file.txt', 'r')
print(practiceFile.read())
```

```
The first line of text
The second line of text
The third line of text
The fourth line of text
I'm bored now, you get the idea
```

2. **<file>.readline()** = Returns one line at a time:

```
practiceFile = open('practice_file.txt', 'r')
print(practiceFile.readline())
```

```
The first line of text
```

3. **<file>.readlines()** = Returns a **list** of lines:

```
practiceFile = open('practice_file.txt', 'r')
print(practiceFile.readlines())
```

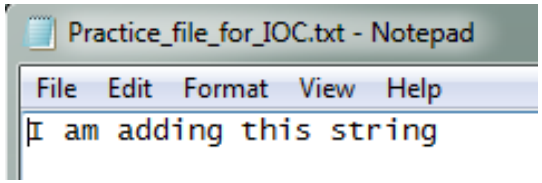
```
['The first line of text\n', 'The second line of text\n', 'The third line of text\n', 'The fourth line of text\n', "I'm bored now, you get the idea\n"]
```

write() function

- There are two similar in-built functions to write to a file:

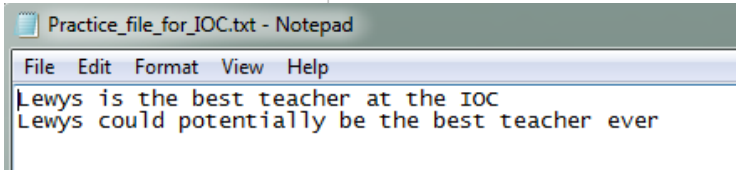
1. **<file>.write()** = Writes a specified sequence of char to a file

```
practiceFile = open('Practice_file_for_IOC.txt', 'w')
practiceFile.write('I am adding this string')
```



1. **<file>.writelines()** = Writes a **list** of strings to a file:

```
testList = ['Lewys is the best teacher at the IOC\n', 'Lewys could potentially be the best teacher ever\n']
practiceFile = open('Practice_file_for_IOC.txt', 'w')
practiceFile.writelines(testList)
```



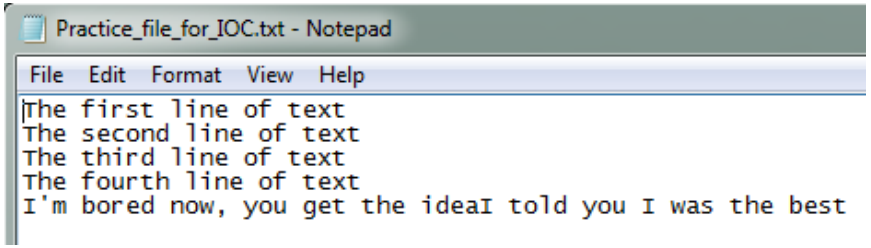
Important: Using the write() or writelines() function will **overwrite** anything contained within a file, if a file of the same name already exists in the working directory.

append() function

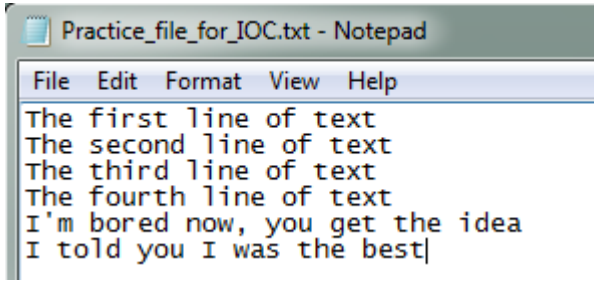
- If you do not want to overwrite a file's contents, you can use the append() function.

- To append to an existing file, simply put 'a' instead of 'r' or 'w' in the open() when opening a file.

```
practiceFile = open('Practice_file_for_IOC.txt', 'a')
testLine = '\nI told you I was the best'
practiceFile.write(testLine)
```



without \n



with \n

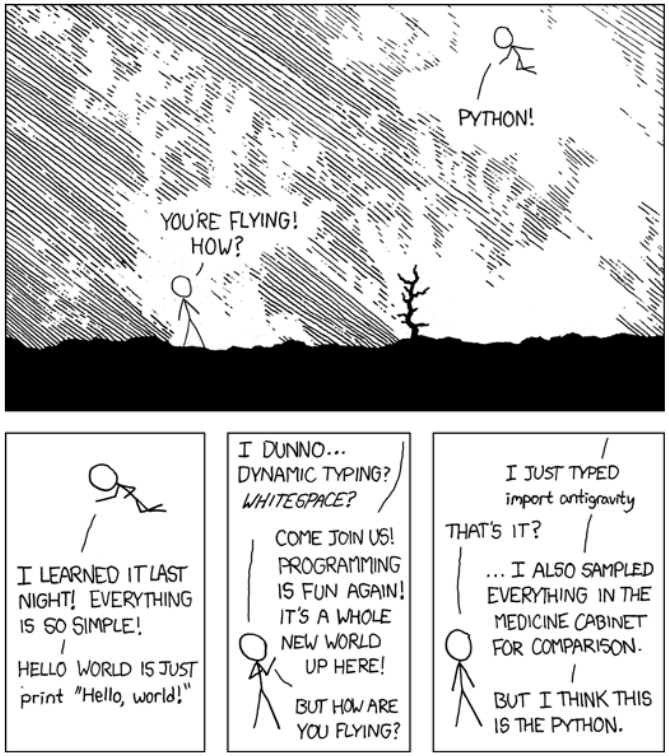
A word on import

- To use a **package** in your code, you must first make it accessible.
- This is one of the features of Python that make it so popular.

```
import datetime
current_time = datetime.datetime.now()
print(current_time)
```

- There are **pre-built** Python packages for pretty much everything.

```
import antigravity
```



Next

1. **Tutorial for basic Python**

   (ELEG4701_Lab_02_Tutorial.docx)

2. **Tutorial for OOP (Object-Oriented Programming)**

   (python-object-oriented-programming.pdf)

3. **Finish Lab sheet 2**

   - When you finished a task session -
   - Raise your hand, and ask the TA to check your codes & and results
   - Submit the Lab Sheet to any TA before you leave
   - Please try **NOT** to use AI tools for this lab sheet (TA will judge)