pythonGeneral

> learn & prepare for PCAP

> every day learn, read, write and revrite code

---

python has dynamic typing

- meaning I can reassign variable to different data type later in code
    - `dogs = 2 <more code> dogs = ["franki", "jessie"]`

its easy for programmer because dont have to declare data type each time you declare something

- it double edge sword because you can hit bugs for unexpected data types
- you can use build in type function **type()** to quickly check the type of any object
- *IN C++* if you declare *dogs* as integer + value you can not change that later in code

---

ASCII vs UTF-8

ASCII is a subset of Unicode.

Unicode is a character encoding standard that can represent almost all characters from all writing systems in the world. It includes a vast collection of characters, symbols, and glyphs that can be used to represent text in any language.

ASCII (American Standard Code for Information Interchange) is a subset of Unicode, which only includes 128 characters that represent the most common Latin letters, digits, and punctuation marks used in the English language.

Unicode builds upon ASCII and includes additional characters from various scripts, including Latin, Greek, Cyrillic, Arabic, Hebrew, Chinese, Japanese, and many others. So, while ASCII only includes a small subset of characters, Unicode includes all ASCII characters and many more.

UTF-8 is an abbreviation for Unicode Transformation Format – 8 bits.

The “8” here means 8-bit blocks are used to represent a character.
UTF-8 is the most commonly used encoding format for Unicode characters.
So, simply speaking, Unicode is a character set and UTF-8 is an encoding format.
With a character set, a character is translated to a decimal number.

ASCII is an abbreviation for American Standard Code for Information Interchange
It is the first character set and character encoding standard for electronic communication.
ASCII contains 128 characters which contain the lower and upper case English letters (a-zA-Z),
the numbers from 0–9, and some special characters.
ASCII is subset of UTF-8

Unidecode is a library that can be used to translate Unicode characters to “approximate” ASCII counterparts. For example, Ä => , Å=> A and Ö=> O.

decompile:

```
#install uncompile if not installed
pip install uncompyle6

uncompyle6 <what_they_dont_wants_U2c>   #std output
uncompyle6 -o . <what_they_dont_wants_U2c>
```

__ dunders __ [double underscores methods]

- any identifier that starts and ends with two underscores (such as `__name__`, `__init__`, `__str__`, etc.) is called a dunder (short for "double underscore") or magic method.
- There are many dunder methods but most commonly used:

```
__init__:
# This is the constructor method that is called when an object of a class
is created.
# Used to initialize the objects attributes.

__str__:
# This method returns a string representation of an object.
# It is used when you call the str() function on an object or when the
object is printed.

__repr__:
# This method returns a string representation of an object that can be used
to recreate the object.
# It is used when you call the repr() function on an object.

__len__:
# This method returns the length of an object. It is used when you call the
len() function on an object.

__getitem__:
# This method is used to get an item from a sequence (such as a list or a
tuple) or a mapping (such as a dictionary) using square bracket notation.

__setitem__:
# This method is used to set an item in a sequence or a mapping using
square bracket notation.

__delitem__:
# This method is used to delete an item from a sequence or a mapping using
square bracket notation.

__call__:
# This method allows an object to be called like a function.

__getattr__:
# This method is called when an attribute that does not exist is accessed
on an object.
# It allows you to define custom behavior for accessing attributes.

__setattr__:
# This method is called when an attribute is set on an object.
# It allows you to define custom behavior for setting attributes.
```

UNDERSCORES:

Underscores are used in Python to indicate various naming conventions and to give special meanings to certain variables and methods.

1. **Single leading underscore:**
    - This convention is used to indicate that a variable or method is intended to be used as a **private member of a class or module**.
    - For example, `_my_private_variable` or `_my_private_method()`.
    - However, this is just a convention and doesn't actually prevent other code from accessing the variable or method.
2. **Single trailing underscore:**
    - This convention is used to avoid naming conflicts with Python keywords or built-in functions.
    - For example, `class_` or `print_`.
3. **Double leading underscore:**
    - This convention is used to implement **name mangling in Python**
    - Name mangling is a technique used to make a variable or method private by adding the class name to its name to avoid naming conflicts.
    - For example, `__my_private_variable` will be converted to `_classname__my_private_variable` when accessed outside the class.
    - This makes it harder to accidentally modify or access the variable or method from outside the class.
4. **Double leading and trailing underscore**
    - This convention is used for special methods or attributes in Python.
    - These methods have a special meaning in Python and are used to implement specific behaviors in classes.
    - For example, `__init__()` is a special method used to **initialize an instance** of a class,
    - and `__doc__` is a special attribute used to access the documentation string of a class or function.
5. **Single underscore**
    - This convention is used as a throwaway variable, indicating that the variable is not going to be used.
    - For example, `_, x = some_function()`, where the first value returned by some_function() is not needed.
    - the single underscore character (_) can also be used as a *variable name or as a placeholder for unused variables or values* for example:

```
# the underscore is used as a placeholder variable.
for _ in range(10):
    # do something
# it is often used when you don't need the value of the current iteration
in a loop. Since the underscore variable is not used within the loop body,
# it tells the reader that the value is not important and can be ignored.

# in normal loop we iterate 10 times, and value of i is used for something:
for i in range(10):
    print(i)
   ...:
0
1
2
3
4
5
6
7
8
9

# but once you place underscore i is not used:
for _ in range(10):
    print(i)
   ...:
9
9
9
9
9
9
9
9
9
9
```

init.py

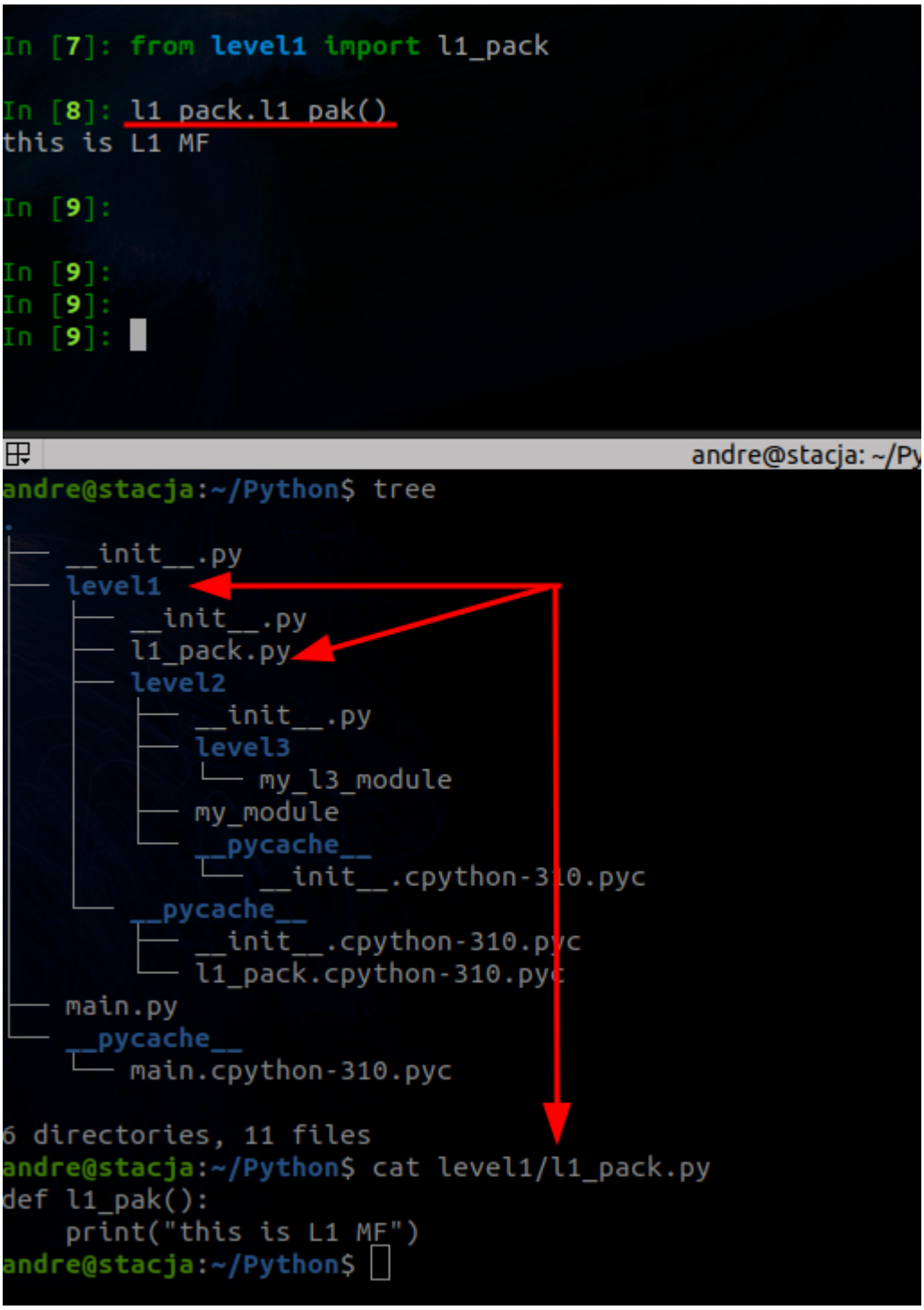
- key .py file is `__init__`.py
- this file marks directory and lets python know that all .py scripts should be threated as a package of modules to import
- so if we want to have more, multiple directories we need this file to be in every of them
- example above where all files in same dir but what if we have subdirectories ?
    - in each subdirectory we need `__inint__` file

Python defines two types of packages, regular packages and namespace packages.

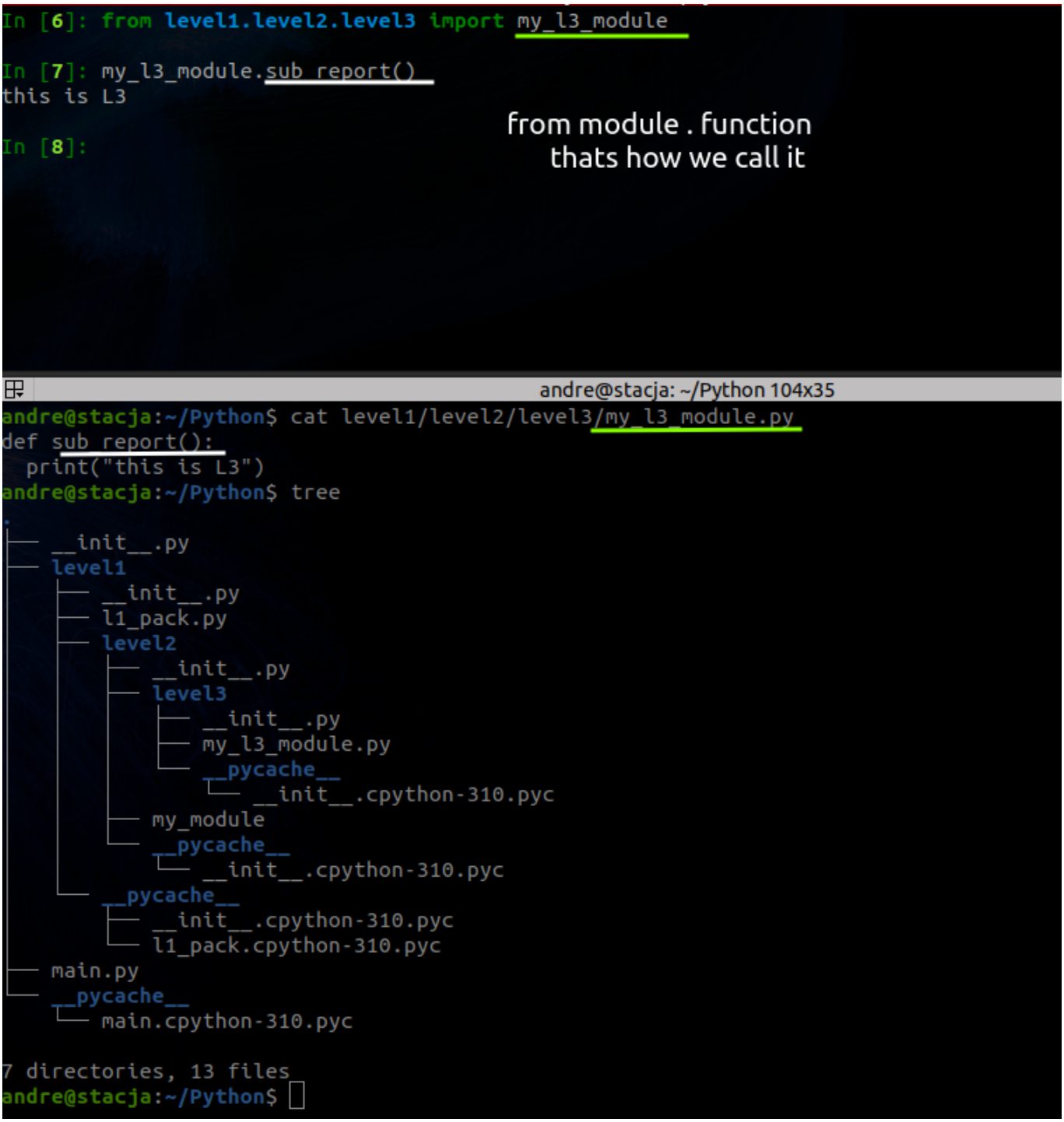
Regular packages are traditional packages as they existed in Python 3.2 and earlier. A regular package is typically implemented as a directory containing an `__init__`.py file.

When a regular package is imported, this `__init__`.py file is implicitly executed, and the objects it defines are bound to names in the package’s namespace.

The `__init__`.py file can contain the same Python code that any other module can contain, and Python will add some additional attributes to the module when it is imported.

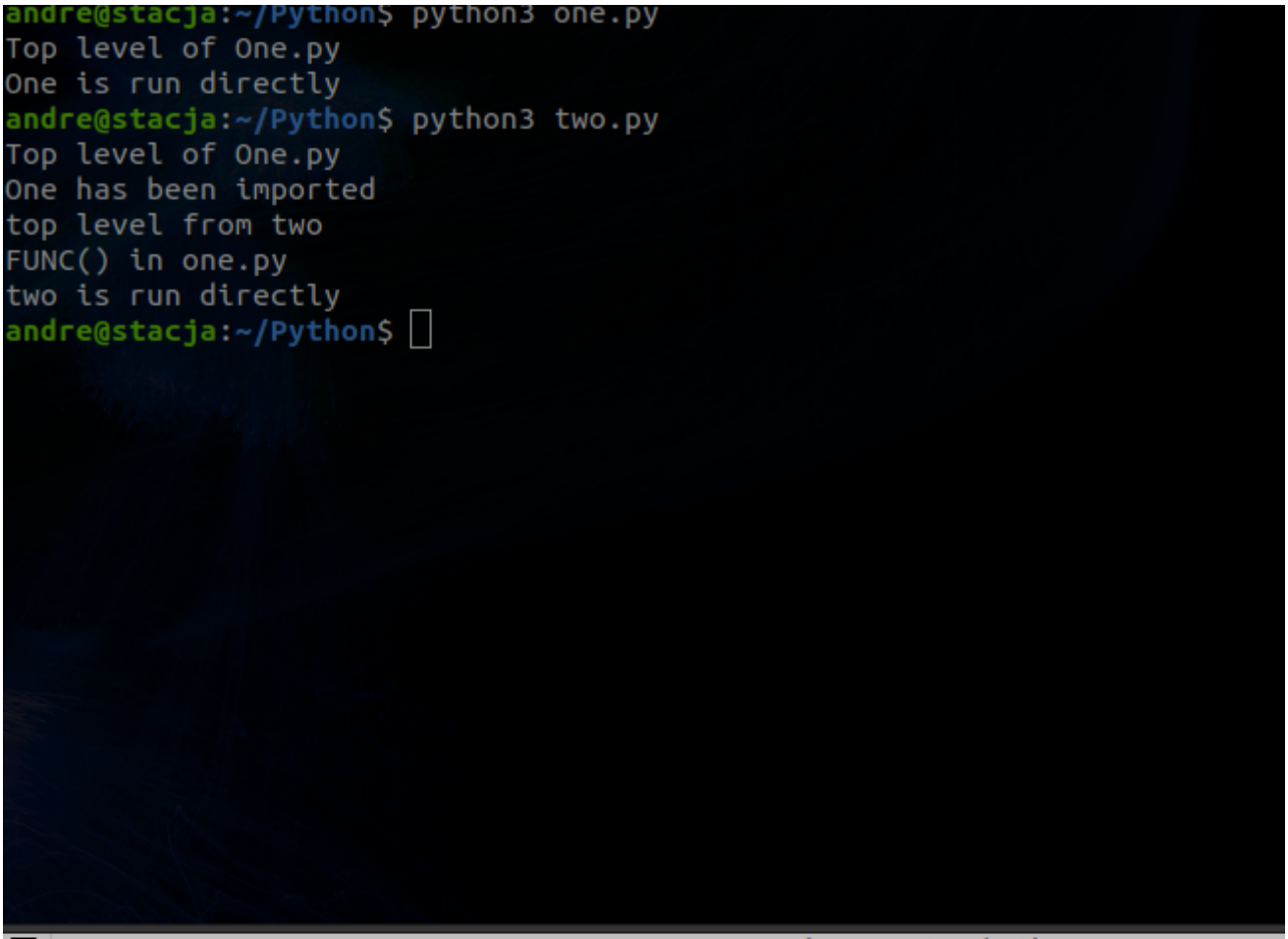

In [7]: from level1 import l1_pack
In [8]: l1_pack.l1_pak()
this is L1 MF
In [9]:
In [9]:
In [9]:
In [9]:
andre@stacja: ~/Py
andre@stacja:~/Python$ tree
.
├── __init__.py
├── level1
│ ├── __init__.py
│ ├── l1_pack.py
│ ├── level2
│ │ ├── __init__.py
│ │ ├── level3
│ │ │ └── my_l3_module
│ │ ├── my_module
│ │ └── __pycache__
│ │ └── __init__.cpython-310.pyc
│ └── __pycache__
│ ├── __init__.cpython-310.pyc
│ └── l1_pack.cpython-310.pyc
├── main.py
└── __pycache__
└── main.cpython-310.pyc
6 directories, 11 files
andre@stacja:~/Python$ cat level1/l1_pack.py
def l1_pak():
print("this is L1 MF")
andre@stacja:~/Python$

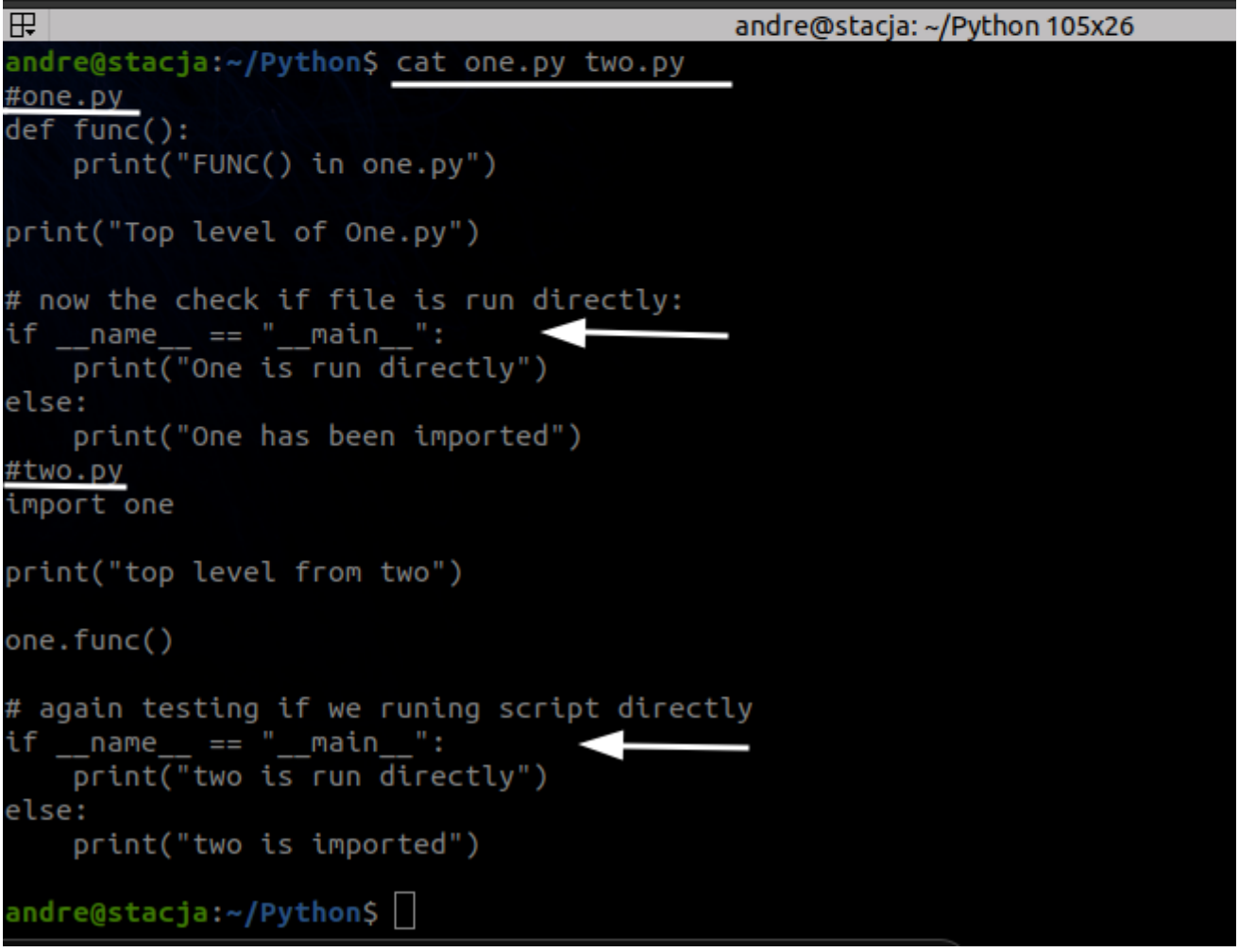
- same like directory structure in linux:
  - `directory/file BUT here MODULE.PACKAGE`



`__name__ __main__`

- sometimes when you import from module you would like to know wheter a module function is being used as import
- or if you are using the oryginal .py file of that module
- `if __name__ == __main__` is usualy at the end of larger script files
  - it is a check if we are running the script
  - it means you run the file, the file that has not been imported
  - `if __name__ == "__main__"` means we runnign whole script, if python can find the name of the file lol


andre@stacja:~/Python$ python3 one.py
Top level of One.py
One is run directly
andre@stacja:~/Python$ python3 two.py
Top level of One.py
One has been imported
top level from two
FUNC() in one.py
two is run directly
andre@stacja:~/Python$


andre@stacja: ~/Python 105x26
andre@stacja:~/Python$ cat one.py two.py
#one.py
def func():
print("FUNC() in one.py")
print("Top level of One.py")
# now the check if file is run directly:
if __name__ == "__main__":
print("One is run directly")
else:
print("One has been imported")
#two.py
import one
print("top level from two")
one.func()
# again testing if we runing script directly
if __name__ == "__main__":
print("two is run directly")
else:
print("two is imported")
andre@stacja:~/Python$

In Python, `__main__` is the name of the default module that is executed when a Python script is run from the command line.

It is also the name of the top-level script environment that Python creates when a module is run as the main program.

When a Python script is run, the interpreter first initializes the environment and sets up some special variables, including `__name__`. If the script is being run as the main program, `__name__` is set to the string "`__main__`". If the script is being imported as a module into another script, `__name__` is set to the name of the module.

This distinction is useful for writing scripts that can be used both as stand-alone programs and as modules in larger programs. By checking the value of `__name__`, a script can determine whether it is being run as the main program or imported as a module, and can take different actions in each case.

For example, a script might define a function or class that can be imported and used by other scripts, but also include some code at the bottom of the script that runs a test of the function or class if the script is being run as the main program. In this case, the test code would be enclosed in an if `__name__ == "__main__"`: block, so that it is only executed if the script is being run directly and not when it is imported as a module.

pycache

A cache is something that keeps a copy of stuff in case you need it again, to save you having to go back to the original

When you run a program in Python, the interpreter compiles it to bytecode first (this is an oversimplification) and stores it in the **pycache** folder.

If you look in there you will find a bunch of files sharing the names of the .py files in your project's folder, only their extensions will be either .pyc or .pyo. These are bytecode-compiled and optimized bytecode-compiled versions of your program's files, respectively.

As a programmer, you can largely just ignore it... All it does is make your program start a little faster. When your scripts change, they will be recompiled, and if you delete the files or the whole folder and run your program again, they will reappear (unless you specifically suppress that behavior).

Python lexis

The lexicon or "lexis" of a programming language refers to its vocabulary, i.e., the set of keywords, identifiers, literals, and symbols that are used to write programs in that language.

In Python, the lexicon includes keywords such as:

- if,
- else,
- while,
- for,
- def,
- class,
- import,
- from,
- as,

As well as built-in functions such as:

- print(),
- input(),
- len().

Python also has several built-in data types, including numbers, strings, lists, tuples, dictionaries, and sets.

In addition to the built-in vocabulary, Python also allows users to define their own identifiers, such as variable names, function names, and class names. These user-defined identifiers become part of the lexicon of any program that uses them.

KEYWORDS TO REMEMBER:

Abstraction

Abstraction is the process of hiding the real implementation of a function from the user and emphasizing only on how to use the function.

encapsuation

Encapsulation is restricting access to methods and variables based on their requirement.

- When a variable name is defined starting with __ (2 underscores), it is a private variable.
- private variable and its value cannot be accessed nor set outside the class.
- This concept of public and private variables is related to encapsulation.

Polymorphism

Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class.

inheritance

Iinheritance, the child class inherits the methods from the parent class.

__bases __

`__bases__` property is a special attribute in Python that is used to access the tuple of base classes of a class. When you create a new class in Python, you can specify one or more base classes from which your new class inherits attributes and methods. The `__bases__` attribute provides a way to access the tuple of base classes that your class inherits from.

`__bases__` property holds the information about the immediate super class of the class. It is not aware of any other hierarchy. Also, it contains the super classes in a tuple and NOT a list.

Here's an example to illustrate how the `__bases__` attribute works:

```
class Animal:
    def speak(self):
        print("I am an animal")

class Dog(Animal):
    def bark(self):
        print("Woof!")

d = Dog()
d.speak()  # Output: "I am an animal"
d.bark()   # Output: "Woof!"

print(Dog.__bases__)  # Output: (<class '__main__.Animal'>,)

# Two classes are defined: Animal and Dog.

# The Dog class inherits from the Animal class, which means that it
automatically inherits the speak method from Animal

# We then create an instance of Dog called d, and we can call both the
speak and bark methods on it.

# Using the __bases__ attribute to access the tuple of base classes of the
Dog class, which in this case is (<class '__main__.Animal'>,). This tuple
contains a single element, which is the Animal class, indicating that Dog
inherits from Animal.
```

global

When used inside a function, the global keyword is used to indicate that a variable defined inside the function should be treated as a global variable, i.e., it should be **accessible from outside the function**.

For example:

```
x = 10

def my_func():
    global x
    x = 20
    print(x)

my_func()
print(x)      # prints 20
x             # prints 20
```

assert:

The assert keyword is used to test if a given condition is true, and if it is not, it raises an exception. Here's an example:

```
# assert <condition>, <optional error message>
x = 10
y = 5

assert x > y, "x is not greater than y"
# This code will raise an AssertionError because the condition x > y is not
true.

# we can also assert type
assert type(my_list) == list, "parameter must be a type of list"
assert len(my_list), "input list must not be empty"
```

try:

The try keyword is used to enclose a block of code that may raise an exception. If an exception is raised within the try block, the code will jump to a except block to handle the exception. Example:

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Error: division by zero")
# This code will catch the ZeroDivisionError that would be raised by trying
to divide 10 by 0, and it will print an error message instead of crashing.
```

raise:

The raise keyword is used to explicitly raise an exception. Here's an example:

```
x = -1

if x < 0:
    raise ValueError("x cannot be negative")
# This code will raise a ValueError exception with the message "x cannot be
negative" if x is less than 0.
```

yield

In Python, **yield** is used in a function to create a **generator object**.

A generator is a special type of iterator that allows you to iterate over a sequence of values without generating the entire sequence at once. Instead, the values are generated on-the-fly as you iterate over the generator.

When you use yield in a function, the function becomes a generator function.

When the generator function is called, it returns a generator object that you can iterate over using a for loop or other iteration methods.

Example:

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1

for num in count_up_to(5):
    print(num)
1
2
3
4
5
# the count_up_to() function is a generator function that generates a
sequence of integers from 1 to n.
# The yield keyword is used to return each integer one at a time, as the
generator is iterated over.
# The for loop iterates over the generator, printing each integer in the
sequence as it is generated.
```

The main advantage of using a generator instead of a list is that generators are more memory-efficient. Because they generate the sequence on-the-fly, they don't need to store the entire sequence in memory at once. This makes them ideal for iterating over very large or infinite sequences of values.

hasattr()

hasattr() is a built-in Python function, not a method. It is used to check whether an object has an attribute with a given name.

The syntax of hasattr() is as follows:

```
hasattr(object, attribute_name)

# object is the object you want to check for the presence of the attribute,
# and attribute_name is a string representing the name of the attribute you
want to check for.
```

The function returns True if the object has the specified attribute, and False otherwise.

Here's an example of how to use hasattr():

```
class MyClass:
    x = 5

obj = MyClass()

if hasattr(obj, 'x'):
    print("obj has attribute 'x'")
else:
    print("obj does not have attribute 'x'")
obj has attribute 'x'

# hasattr() is used to check whether the object obj has an attribute named
x. Since the attribute exists in the MyClass class, which obj is an
instance of, the function returns True, and the message "obj has attribute
'x'" is printed to the console.
```

hasatr() can also be used to check for attributes from sublcasses, example:

```
class Class1:
    x = 1

class Class2(Class1):
    y = 4

class Class3(Class2):
    z = 0

# Check if Class3 has an attribute from Class2
if hasattr(Class3, 'y'):
    print('Class3 has attribute y from Class2')
```

```
# Check if Class3 has an attribute from Class1
if hasattr(Class3, 'x'):
    print('Class3 has attribute x from Class1')

# class3 inherits from class2, which in turn inherits from class1.
# When an attribute is accessed on an object of class3, Python will first
check if the object has the attribute defined directly on it.
# If not, it will check if the attribute is defined in the class of the
object, and if not, it will check if it is defined in the superclass of the
class and so on until it reaches the top-level object class.

# output:
Class3 has attribute y from Class2
Class3 has attribute x from Class1
```

chr() ord()

chr() and **ord()** are two built-in Python functions that are used for working with characters and their corresponding Unicode code points.

chr() takes an integer representing a Unicode code point and returns the corresponding character.

```
chr(65)
"A"
# since the Unicode code point for 'A' is 65.
# chr() can also take a string representing an integer in the range 0-
1114111 (which is the maximum Unicode code point) and returns the
corresponding character.
```

ord() takes a character and returns its corresponding Unicode code point as an integer.

```
ord('A')
65
# which is the Unicode code point for 'A'.
# ord() can also take a string containing a single character and returns
the corresponding Unicode code point.
```

These functions are often used together to convert characters to their corresponding code points and vice versa. For example,

```
ord(chr(65))
65
# 65 which is the Unicode code point for 'A',
chr(ord('A') + 1)
'B'
# which is the character corresponding to the Unicode code point 66.
```

PyPi

- is a repository for open-source third-party python packages
- with PIP you install from PyPi
- so for example if we want to use **colorama** we install it with pip
- `pip install colorama` and now we can use it in our scripts

so withy PyPI we can install external modules and packages

IMPORT:

Import modules using the **import** statement.

```
# Having module named my_module.py that contains some functions and
variables.
# To import this module, you can use the following syntax:

import my_module
# This will import the my_module module and make all of its functions and
variables available in your program.
# You can access these functions and variables using the my_module. prefix.
For example:

import my_module

result = my_module.add(2, 3)
print(result)
# In this example, the add function from my_module is called using the
my_module. prefix.
```

Another way to import modules is to use the **from** statement. For example:

```
from my_module import add

result = add(2, 3)
print(result)

# In this example, only the add function is imported from the my_module
module.
# This means you can call the add function directly without using the
my_module. prefix.

# Using from can be useful when you only need to use a few functions or
variables from a module.
# However, be careful not to use from too much, as it can make your code
harder to read and maintain.
# Also, be aware that if you use from to import a function or variable, you
won't be able to access any other functions or variables in that module
without importing them separately.
```

***other explanation**

In Python, a module is simply a file containing Python definitions and statements. To use the functions, classes, and variables defined in a module, you need to import it into your current Python script.

There are two styles of import statements in Python:

1. **import module_name**:

- This style of import statement imports the entire module and makes it available under the module's name in your current script.
- You can access the functions, classes, and variables defined in the module using dot notation.
- For example, if you have a module called my_module that contains a function called my_function, you can import the module using the following statement:

```
# import:
import my_module
# access the function using dot notation:
my_module.my_function()
```

2. **from module_name import name**

- This style of import statement allows you to import specific functions, classes, or variables from a module and make them available directly in your current script.
- You do not need to use dot notation to access them.
- For example, if you only need to use the my_function function from the my_module module, you can import it using the following statement:

```
from my_module import my_function
# Then, you can use the function directly in your script without using the
module name or dot notation:
my_function()
```

The main difference between the two styles of import statements is the namespace they create. When you use import module_name, you create a new namespace for the entire module, and you need to use the module name and dot notation to access its contents. When you use from module_name import name, you import only the specific function, class, or variable you need and add it directly to your current namespace.

For example, if you use import my_module, you need to use the module name and dot notation to access its contents: `my_module.my_function()`

But if you use from my_module import my_function, you can use the function directly without the module name or dot notation:`my_function()`

errors | error handling

AttributeError can be defined as an error that is raised when an attribute reference or assignment fails.

One handy way of deciding between value error and type error is by understanding that value error occurs when there is a problem with the content of the object you tried to assign the value to for example the statement int("a") produces a ValueError because the value being provided to int is of the wrong type (Don't associate with TypeError).

One handy way of deciding between value error and type error is by understanding that value error occurs when there is a problem with the content of the object you tried to assign the value to for example the statement int("a") produces a ValueError because the value being provided to int is of the wrong type (Don't associate with TypeError). Now the 'type' of the 'func' function in the question is such that it REQUIRES a value. It is the TYPE (read nature) of this function that it must need a value to work and if we call it simply like func() it violates its nature. Hence a TypeError occurs.

try except else

```
# try:
#   code that might break your program
#   important about incantentions
# except:
#   here what to do if we run into error..
#   print("hey you doing sth wrong here...")
# else:
#   this block of code is what program sopousd to do if run with no errors

try:
  for i in ['a','b','c']:
    print(i**2)
except:
  print("watch out error here")
finally:
  print("all done")
```

try except finally

```
# try:
#   your code here, example opening file:
#   f = open("testfile","w")
#   f.write("write a test line")
# except TypeError:
#   #this line excepts only TypeErrors
# except:
#   thsi will except all other errors
```

```
#   but you want to have code doign sth with this exception
# finally:
#   this will always run, no matter if errors in code or not, it always
runs
try:
  for i in ['a','b','c']:
    print(i**2)
except:
  print("watch out error here")
finally:
  print("all done")
```

```
def ask_for_int():

    while True:
        try:
            result = int(input("Please provide number: "))
        except:
            print("Whoops! That is not a number")
            continue
        else:
            print("Yes thank you")
            break
```

WHILE loop means we will keep doing same stuff again and again until hit break

CONTINUE every time we hit error, we print this message and continue with loop

so until we have errors we continue, break only when its errors free

BREAK keyword is here, so once python gets to this it will break WHILE loop,

testing tools

- when working with other ppl you have to test your new code to make sure the old code didnt break

pylint

- library that looks at your code and reports back possible issues
- it also gives you score of your code up to 10/10 😄

```
pip install pylint
pylint filename.py
```

unittest

- buil-in library will allow you to test your own programs and check you are getting desired outputs
- so you have to:

1. inside your Testing Script import unittest
2. import all other modules / scripts you have been working on

3. create a class for testing (unittest.testcase)
4. and than you have methods to test any situation you can think of, basicaly it checks your code
5.

module is just a .py script used in another script

- packages are collection of moduels
- module is really just .py script..its fancy way of saying here Im using another .py script
- and this is how you actualy write nice scripts / programs: that there are multiple files and you importing stuff from them

```
# file_one with function my_func

# file_two (same directory) and we want to import my_func this is how:
# - from file_one import my_func
# BUT both files are in same directory
```

DATA TYPES:

every object in python has associated data type and data types determine what you can (and can not) do with the object...

data type is basic building block when constructing larger piece of code

Fundamental data types:

- int
  - integers: whole numbers
- float
  - floating point: numbers with decimal point, even when .0 its still an integer ex: 100.0
- str
  - strings: ordered sequence of characeters, strings has double or single quotes so number 200 put in doublequotes *"200" is a string*

data structures:

https://docs.python.org/3/tutorial/datastructures.html#

data structure a bit more specialized than basic data types, because they can hold data types in sequence or mapping,

- list:
  - *ordered sequence of objects*
- dic:
  - can store another data types in *unordered KEY:Value* pairs
- tuples:

    - tup, is *ordered, immutable sequence of objects*
    - they look a lot like lists but are immutable, meaning - you can not change object that is already in that sequence
- bool:
    - booleans, are *logical value of True or False*
- set:
    - sets are *unordered collection of unique objects*
    - None (NoneType - represents absence of value)

How to check the type of object in python?

```
#
# simply use build in type() function
#
type(2)
<class 'int'>
andre = ["handsome","lowing","fit","inteligent"]
>>> andre
['handsome', 'lowing', 'fit', 'inteligent']
type(andre)
<class 'list'>
```

VARIABLES

- pointers to place in memory that stores a value
- and they have a name (identifier)
- variable andre from above stores values, which are my characteristics
- so the values are binded to the name
- variable names should be descriptive (logical right)
- variables are case sensitive
- variables should be lowercase, (maybe only global, all global variables to be in CAPS)
- lower_case_words_separated_by_underscores (snake cases thats how they are called)
- just dont use python keywords: https://runestone.academy/runestone/books/published/py4e-int/variables/variable-names-keywords.html
- can not start with number
- can not have symbols (:!"(){@:"£$%^^&||"}")

```
#
# ASSIGN VALUES TO VARIABLE andre
#
andre = ["handsome","lowing","fit","inteligent"]
#
# MULTIPLE VARIABLE ASSIGNMENT
#
# we can also assign values to multiple variables in one go:
#
```

```
# three  variables: me, you, us
me, you, us = "andre", "anita", "zareczeni i szczesliwi"

# and can perform logic with variable names:
#
print(me, "and", you)
andre and anita

# or example of using variables (english like names):
#
In [20]: my_income = 100
In [21]: tax_rate = 0.4
In [23]: my_taxes = my_income * tax_rate

In [24]: my_taxes
Out[24]: 40.0
```

STRINGS

- ordered sequence of characters
- with indexing that starts at 0 and reverse indexing starting with -1
- you can also slice strings
    - [start:stop:step]
        - start - numerical index of start
        - stop - is the index you go up to but not including
        - step is just the size of jump
- have to print() to call them
- strings are wrapped inside '' or ""

strings comparison

- Python String comparison can be performed using equality (==) and comparison (<, >, !=, <=, >=) operators.
- characters in both strings are compared one by one
- when different characters are found thn their Unicode value is compared

Code Point Range	Class
0 through 31	Control/non-printable characters
32 through 64	Punctuation, symbols, numbers, and space
65 through 90	Uppercase English alphabet letters
91 through 96	Additional graphemes, such as [and \
97 through 122	Lowercase English alphabet letters
123 through 126	Additional graphemes, such as { and |
127	Control/non-printable character (DEL)

string indexing

- with slicing you can reverse the string simply by
  - mystring[::-1] what is does is from all they way from begginigng first : to the end, second : with step of engative one -1 czyli from end lol
- indexing can be done on strings as they are in variable and on strings themselves:
  - "hello"[3] gives l
- starting at index 5 and going all the way to end:
  - "hallo my friend"[5:]

string properties and methds

- IMMUTABLE:
  - means string object doesnt support item assignment
    - you can not change/reassing one/many letters/items in any of the strings:

```
# change S to P in Sam
name = "Sam"
name[0] = "P"
  TypeError: 'str' object does not support item assignment
# achieve than creating new variable:
"P" + name[1:]
  'Pam'
```

- CONCATENATION:

    - adding strings together (like P+slicing to end in above example)
    - or multiplying:

        ```
        "yes" * 10
          yesyesyesyesyesyesyesyesyesyes
        ```

- **STRING**. the dot brings up a list of methods you can use on strings

    - string**.upper()** and now we are screaming

string formatting

.format() method

```
In [35]: a = "andre"

In [36]: b = "best"

In [37]: c = "pro"

In [38]:

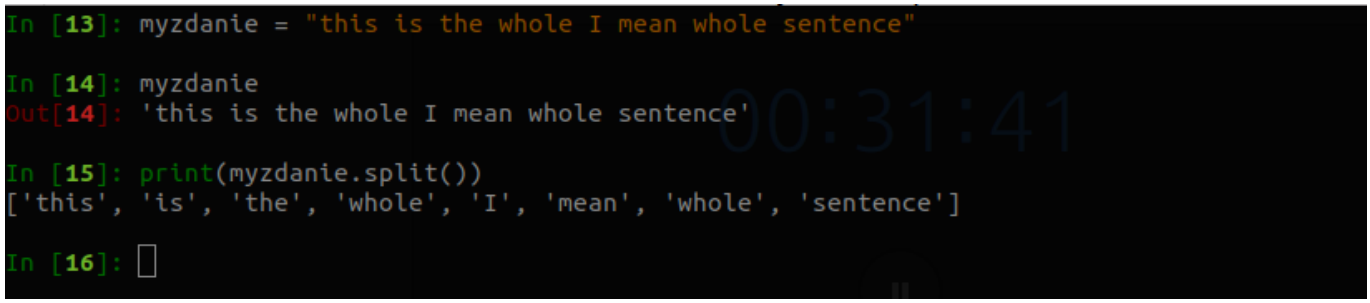
In [38]: print("who is the best hacker? A: {} ! And who is {} ?
{}".format(a,b,a
    ...: ))
who is the best hacker? A: andre ! And who is best ? andre
#
# PLACING ONE THING:
#
logerror = "nvme disk is corrupt"

print("The error we received was: ",
colored(logerror,"red").format(logerror))

The error we received was:  nvme disk is corrupt
```

string split()

- split() is very handy if we want to split whole sentence into individual words



string join()

- join on anything you want, white space ?

```
In [21]: newzdanie = myzdanie.split()

In [22]: newzdanie
Out[22]: ['this', 'is', 'the', 'whole', 'I', 'mean', 'whole', 'sentence']

In [23]: " ".join(newzdanie)#
Out[23]: 'this is the whole I mean whole sentence'

In [24]:
```

string colors

- prerecs:

```
# termcolor installed in ubuntu packages
sudo apt install python3-termcolor
pip3 install termcolor

# and imported in script / ipython
from termcolor import colored
```

```
In [60]: logerror = "nvme disk is corrupt"

In [61]: print("The error we received was: ", colored(logerror,"red").format(logerror))
The error we received was:  nvme disk is corrupt

In [62]:
```

strings lower() function

```
In [32]: st = "Print only the words that start with s in the sentence and print words that starts with capital or norm
    ...: al P"

In [33]: for word in st.split(' '):
    ...:     if word.startswith('s') or word.lower().startswith('p'):
    ...:         print(word)
    ...:     else:
    ...:         continue
    ...:
    ...:
Print
start
s
sentence
print
starts
P

In [34]:
```

if we are looking for either Capital letter or lower case good idea is to use **.lower()** function so it lowers all cases

split

```
split() is used to split a string into a list of substrings based on a
separator. It takes one optional argument sep which is the separator to use
when splitting the string. If sep is not provided, any whitespace
```

```
characters (spaces, tabs, newlines) will be used as the separator. Here's
an example:


sentence = "The quick brown fox jumps over the lazy dog"
words = sentence.split() # split into words using whitespace as separator
print(words)
# Output: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy',
'dog']

filename = "myfile.txt"
parts = filename.split(".") # split into filename and extension
print(parts)
# Output: ['myfile', 'txt']
```

join

```
join() is used to join a sequence of strings (e.g. a list or tuple) into a
single string, using a separator string. Here's an example:

words = ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy',
'dog']
sentence = " ".join(words) # join the words with a space separator
print(sentence)
# Output: 'The quick brown fox jumps over the lazy dog'

filename_parts = ['myfile', 'txt']
filename = ".".join(filename_parts) # join the filename and extension with
a dot separator
print(filename)
# Output: 'myfile.txt'
Note that join() is a method of the separator string, so you call it on the
separator and pass the sequence of strings to be joined as an argument.
```

ARITHMETICS

```
# aughmented assignment in two ways can be done:
#
# -
counter = 0
counter = counter + 1
# albo to samo ale szybciej
counter +=  1

# same with multiplier or other arithmetics:
counter -= 1
counter *= 2
```

```
# there is operator precedence (czyli co jest liczone pierwsze ) called
PEMDAS
#
# P   - parantesis
# E   - exponentiansions (rising to the power of)
# M   -  multiplications
# D   - division
# A   - additions
# S   - substractions
```

FLOATS & INTEGERS

- FLOATs take more space in memory
- you can change objects to FLOATs with float() function
- same with intigers, change FLOAT to INTEGER with int() function,
- when changing to INT you NOT rounding you just ignore the part that is fractional so 3.9999 will be just 3
- FLOATs are aproximations, because they have to be stored as binary fractions and python sometimes have issues with this, (some nubers can not be represented in binary), more on subject read this: https://de.wikipedia.org/wiki/IEEE_754
-

BOOLEANS

- True/False with case sensitive

OPERATORS

- = assignment operator
- == comparison operator
- != not equal
- > = grater than or equla to
- '' string
- "" multi sting
- ALTERNATIVE QUOTATION:
  - use \ before special character:
    - 'Let's code "pandorably"'
  - use tripple quotes:
    - '''within tripple codes I'm able to use anythig "i" want'''
- % operator is the modulus operator, which returns the remainder of a division operation between two numbers. For example:

```
a = 17
b = 5
```

```
c = a % b

print(c) # Output: 2
```

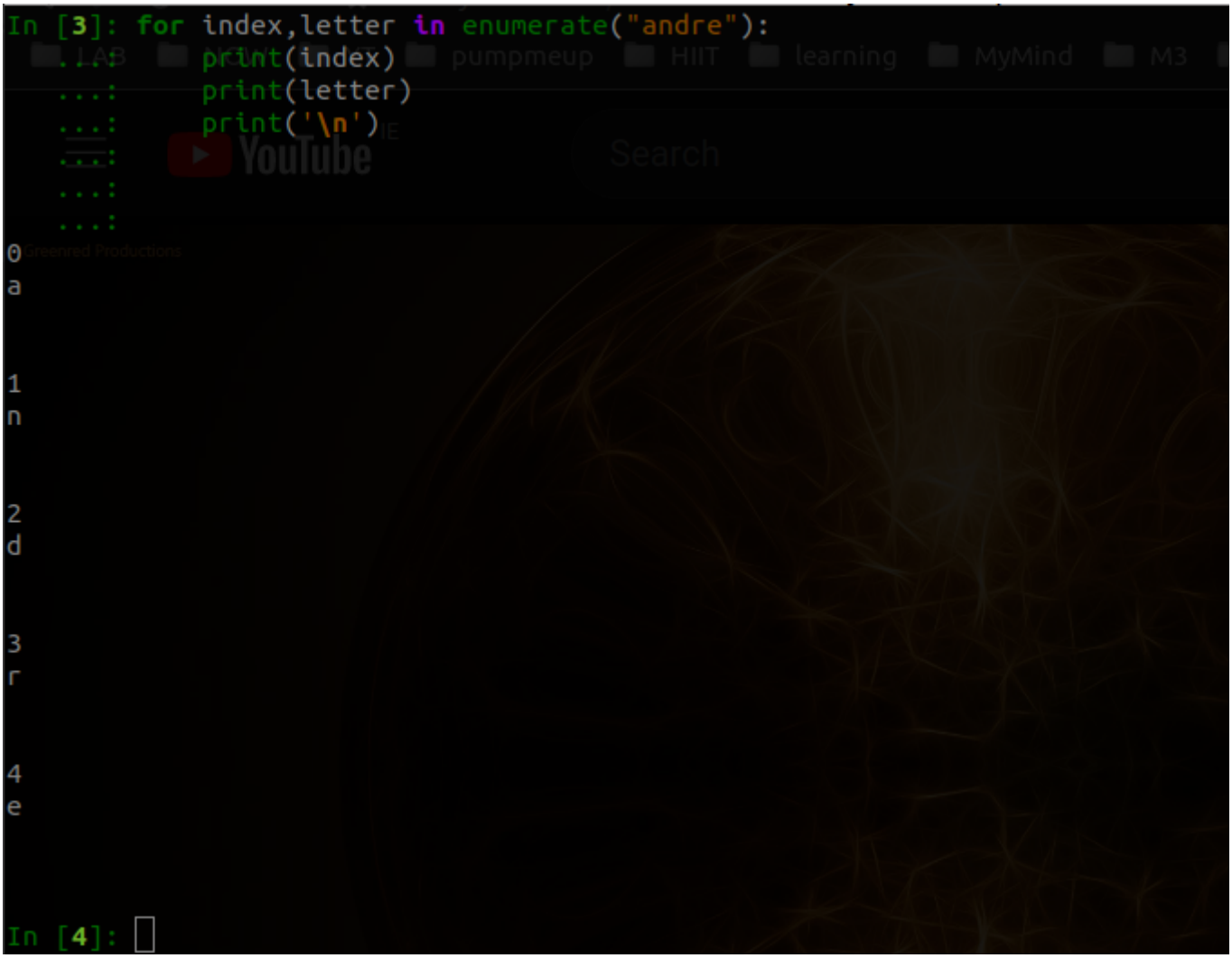
- / operator is the division operator, which performs a regular division operation between two numbers.

Operator	Description	Example
`+`	Addition	`3 + 2` evaluates to `5`
`-`	Subtraction	`3 - 2` evaluates to `1`
`*`	Multiplication	`3 * 2` evaluates to `6`
`/`	Division	`3 / 2` evaluates to `1.5`
`//`	Floor Division	`3 // 2` evaluates to `1`
`%`	Modulus (Remainder)	`3 % 2` evaluates to `1`
`**`	Exponentiation	`3 ** 2` evaluates to `9`
`==`	Equal	`3 == 2` evaluates to `False`
`!=`	Not Equal	`3 != 2` evaluates to `True`
`<`	Less Than	`3 < 2` evaluates to `False`
`>`	Greater Than	`3 > 2` evaluates to `True`
`<=`	Less Than or Equal To	`3 <= 2` evaluates to `False`
`>=`	Greater Than or Equal To	`3 >= 2` evaluates to `True`
`and`	Logical And	`True and False` evaluates to `False`
`or`	Logical Or	`True or False` evaluates to `True`
`not`	Logical Not	`not True` evaluates to `False`
`in`	Membership	`'a' in ['a', 'b', 'c']` evaluates to `True`
`not in`	Negative Membership	`'d' not in ['a', 'b', 'c']` evaluates to `True`
`is`	Identity	`3 is 3` evaluates to `True`
`is not`	Negative Identity	`3 is not 2` evaluates to `True`

OPERATORS

ENUMERATE

- prints tuples of index value + whatever it was going through
- example we want to have index of each letter in word:
- can take any itterable object and it returns index counter and the object itself



ZIP

- zip together two lists

```
#
# having two lists and want to combine them:
#
>>> andre2
[1, 2, 3, 4, 5]
>>> andre
```

```
['fit', 'dedicated', '5AC', 'dedicated', 'handsome', 'fit', 'wolf']
>>>
>>> list(zip(andre,andre2))
[('fit', 1), ('dedicated', 2), ('5AC', 3), ('dedicated', 4), ('handsome',
5)]
>>>
#
# UNPACKING THE VALUES FROM ZIPPED ONES:
#
# 1) assign new names
for andre_adjective, and_number in zip(andre,andre2):
  print(andre_adjective)

# 2) example:
# assign new names to variables that we will place from zipped andre,andre2
>>> for andre_adjective, and_number in zip(andre,andre2):
...   print("andre is {} and {} times".format(andre_adjective,and_number))
# .format() method
...
andre is fit and 1 times
andre is dedicated and 2 times
andre is 5AC and 3 times
andre is dedicated and 4 times
andre is handsome and 5 times
```

IN

- IN operator
- check if object is in list, tuple, dictionary, strings
- returns boolean

The in keyword has two purposes:

The in keyword is used to check if a value is present in a sequence (list, range, string etc.).

The in keyword is also used to iterate through a sequence in a for loop:

MIN / MAX

RANDOM

explain random.seed() why random.seed(1) is always the same ? other examples ?

n Python, random.seed() is a method from the random module that allows you to set the starting point for generating random numbers. The starting point is known as the seed, and the same seed will always produce the same sequence of random numbers.

```
import random

# Generate a random number
num1 = random.random()
print(num1)

# Set the seed to 1
random.seed(1)

# Generate another random number
num2 = random.random()
print(num2)

# Generate another random number with the same seed
num3 = random.random()
print(num3)

# In this example, we first generate a random number using random.random().
This will generate a random number between 0 and 1.

# We then set the seed to 1 using random.seed(1). This means that any
subsequent calls to random methods (e.g. random.random(), random.randint(),
etc.) will produce the same sequence of random numbers as long as the seed
remains the same.

# We then generate another random number using random.random(), which
produces a different number than the first one because we set the seed to
1.

# Finally, we generate another random number using random.random() with the
same seed, and this produces the same number as the second one because we
used the same seed.

# This predictability can be useful in certain cases, such as in scientific
simulations, where reproducibility is important.

# Another example is to use random.seed() to shuffle a list in a
deterministic way. Here's an example:

import random
```

```
# Create a list of numbers
numbers = [1, 2, 3, 4, 5]

# Shuffle the list with a seed of 1
random.seed(1)
random.shuffle(numbers)
print(numbers)

# Shuffle the list again with the same seed
random.seed(1)
random.shuffle(numbers)
print(numbers)

# In this example, we first create a list of numbers. We then shuffle the
list using random.shuffle() with a seed of 1. This produces a random
ordering of the list. We then shuffle the list again with the same seed,
which produces the same ordering as the first shuffle.

random.random() generates a random float number between 0 and 1 (inclusive
of 0, but exclusive of 1), according to the documentation.

If you want to generate random numbers within a different range, you can
use other methods from the random module, such as randint(), uniform(), or
gauss(). For example, random.randint(a, b) generates a random integer
between a and b (inclusive of both a and b).
import random

# Generate a random integer between 1 and 10 (inclusive)
num1 = random.randint(1, 10)
print(num1)
2

# Generate a random float between 0 and 1
num2 = random.random()
print(num2)
0.4594964569679978

# Generate a random float between -1 and 1
num3 = random.uniform(-1, 1)
print(num3)
0.6965599747214852

# Generate a random number from a Gaussian distribution with mean 0 and
standard deviation 1
num4 = random.gauss(0, 1)
print(num4)
-1.1447287731483977
```

INPUT

- always takes as strings

```
In [90]: a = input("please provide a number ")
please provide a number 3

In [91]: a
Out[91]: '3'

In [92]: type(a)
Out[92]: str

In [93]:
```

- so what do we do to have input exaclty like we want ?
    - 1. make sure the data type is what we want
    - 2. check if input is within limits, or within choices we want

user input validation examples

1. if type == init

2. isdigit()

```
In [13]: def get_digit():
    ...:
    ...:     choice = "wrong"
    ...:
    ...:     while choice.isdigit() == False:
    ...:
    ...:         choice = input("Please provide a number 0-10")
    ...:
    ...:         if choice.isdigit() == False:
    ...:
    ...:             print("This is not a digit!")
    ...:     return int(choice)
    ...:

In [14]: get_digit()
Please provide a number 0-101
Out[14]: 1

In [15]: get_digit()
Please provide a number 0-10100
Out[15]: 100

In [16]: get_digit()
Please provide a number 0-10two
This is not a digit!
Please provide a number 0-10three
This is not a digit!
Please provide a number 0-103
Out[16]: 3

In [17]:
```

now same example but with more logic and checks:

```
In [21]: def get_digit():
    ...:
    ...:     # DEFIND VARIABLES:
    ...:     choice = "wrong"
    ...:     acceptable_range = range(0,10)
    ...:     within_range = False
    ...:
    ...:
    ...:     # CONDITIONS we are checking for:
    ...:
    ...:     # need to check if this is a digit, so keep asking until input is digit
    ...:     # or keep asking until within_range is false
    ...:     while choice.isdigit() == False or within_range == False:
    ...:         # so until those two ^ are not meet we are askikng this:
    ...:
    ...:         # THIS IS OUR DIGIT CHECK
    ...:         choice = input("Please provide a number 0-10: \n")
    ...:
    ...:         if choice.isdigit() == False:
    ...:
    ...:             print("This is not a digit!")
    ...:
    ...:         # RANGE CHECK:
    ...:         # so first we need to make sure that the choice is digit is true:
    ...:         if choice.isdigit() == True:
    ...:
    ...:             # now make sure we are within range:
    ...:             # ...and because we know isdigit is true now so we are passing digit
    ...:             if int(choice) in acceptable_range:
    ...:                 # if the number is within range we are updating it to be true:
    ...:                 within_range = True
    ...:
    ...:                 # if however we are still not in range, we need to update it to false
    ...:             else:
    ...:                 print("you are out of acceptable range 0-10 ")
    ...:                 within_range = False
    ...:
    ...:     return int(choice)
    ...:
    ...:

In [22]: get_digit()
Please provide a number 0-10:
11
you are out of acceptable range 0-10
Please provide a number 0-10:
two
This is not a digit!
Please provide a number 0-10:
five
This is not a digit!
Please provide a number 0-10:
10
you are out of acceptable range 0-10
Please provide a number 0-10:
9
Out[22]: 9
```

3. try: validate = int(userinput) except valueerror: print thats not intiger

4. INTERACTION WHERE USER PROVIDES VARIABLE WHICH CAN BE CHANGED BY HIM

- like we can use it for RCA ?
- user provides *timeframe* which we can use to search logs
- and that timeframe can be changed by user
- do kazdego zadania jest zrobiona funkcja

1. DISPLAY FUNCTION:

```
#
# fist declare your game list variable
#
game_list = [0,1,2]

#
# than create function that will display that variable
#
def display_game(game_list):
  print("Here is current list: ")
  print(game_list)
```

2. FUNCTION TO GET POSITION CHOICE

```
#
# always have to have INITIAL choice, here we set it as WRONG
#
def position_choice():

  choice = "wrong"

  # here we know that the choice must be either 0,1 or 2
  # in order to close off this wile loop we need to receive this three
things
  while choice not in ['0','1','2']:

    choice = input("Pick a position 0,1,2: ")

    # czyli teraz if the choice is not in 0,1,2 we will print sorry provide
the valid one
    if choice not in ['0','1','2']:
      print("Sorry, invalid choice!")

  # while loop is going over and over until we get correct choice
  # and when we get correct than we will return it
  return int(choice)
```

3. REPLACEMENT FUNCTION

```
#
# this function is after player has choosen correctly !
# - not player chooses replacement value
# - this one takes in two parameters: game_list and position return from
previous function
def replacement_choice(game_list,position):

  user_placement = input("Type a string to place at position: ")

  # now we grab game list at the position previously passed and we make it
same as user_placement
  game_list[position] = user_placement

  # and of course you have to return that gamelist
  return game_list
```

```
In [10]: def replacement_choice(game_list,position):
   ...:     user_placement = input("Type a string to place at position: ")
   ...:
   ...:     # now we grab game list at the position previously passed and we make it same as user_placement
   ...:     game_list[position] = user_placement
   ...:
   ...:     # and of course you have to return that gamelist
   ...:     return game_list
   ...:

In [11]: replacement_choice(game_list,1)
Type a string to place at position: newOne
Out[11]: [0, 'newOne', 2]

In [12]:
```

4. NOW FUNCTION WHERE YOU PICK IF YOU WANT TO KEEP PLAYING

```
#
#
#
def gameon_choice():
  # again we need to specify wrong choice first
  choice = "wrong"
  # and we will ask Y or N until we get the correct answear
  while choice not in ['Y', 'N']:
    choice = input("pick Y or N ")

    # if wrong choice we will print its wrong
    if choice not in ['Y', 'N']:
      print("only Y or N accepted ")

  # now we returning one choice either Y or N so we need if statement here
  if choice =='Y':
    return True
  else:
    return False
```

```
In [10]: def replacement_choice(game_list,position):
   ...:     user_placement = input("Type a string to place at position: ")
   ...:
   ...:     # now we grab game list at the position previously passed and we make it same as user_placement
   ...:     game_list[position] = user_placement
   ...:
   ...:     # and of course you have to return that gamelist
   ...:     return game_list
   ...:

In [11]: replacement_choice(game_list,1)
Type a string to place at position: newOne
Out[11]: [0, 'newOne', 2]

In [12]:
```

5. NOW PUT ALL TOGETHER

```
# first declare that game on is true
game_on = True
```

```
# and declare our list
game_list = [1,2,3]
# while gameon is true we display the game with current game list

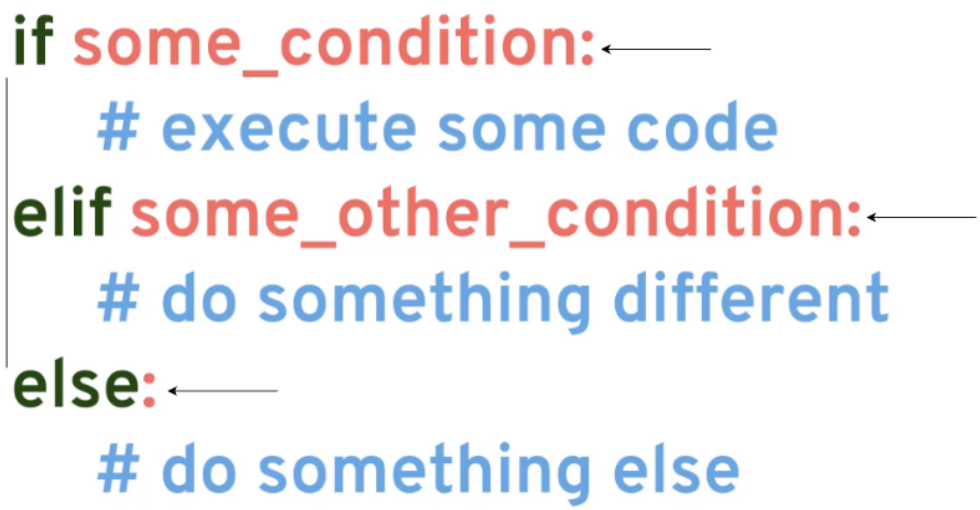
while game_on: # since we have boolean we dont have to do "while game_on ==
True:"
  display_game(game_list)
  # we want to hvae display position function now where
  position = position_choice()
  # so once postition_choice function is called we know that player has
provided position
  # we need to rewrite that position and display new game_list
  game_list = replacement_choice(game_list, position) #replacement_choice
function takes in game list and postition so we need to specify it here
  # and since replacement_choice returns game_list we just reassigned all
this to game_list

  # finnaly we display game with updated game_list
  display_game(game_list)

  # with while loop this would go for ever so we need to give user choice
to break it
  # using gameon choice function
  game_on == gameon_choice()
```

STATEMENTS

IF ELSE



FOR LOOPS

for loops iterate over something

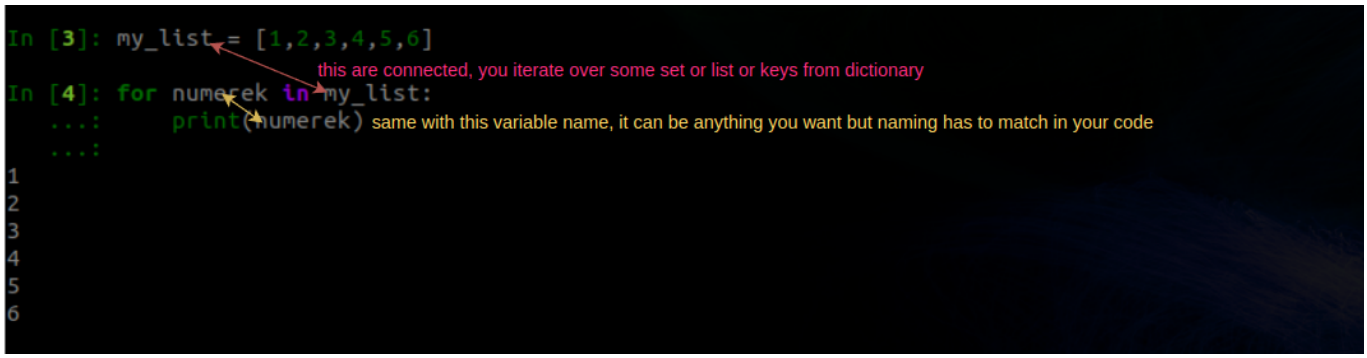
iterate meaning we can iterate over every element in the object, it could be:

- every element in a list
- or every character in a string,
- for every line in document,
- or iterate every key in dictionary

and for loops are used to iteration

```
my_iterable = [1,2,3]
for item_name in my_iterable:
    print(item_name)
```

for every item in items name we execute code (print item in this example)



with tuple unpacking we have acces to individual items

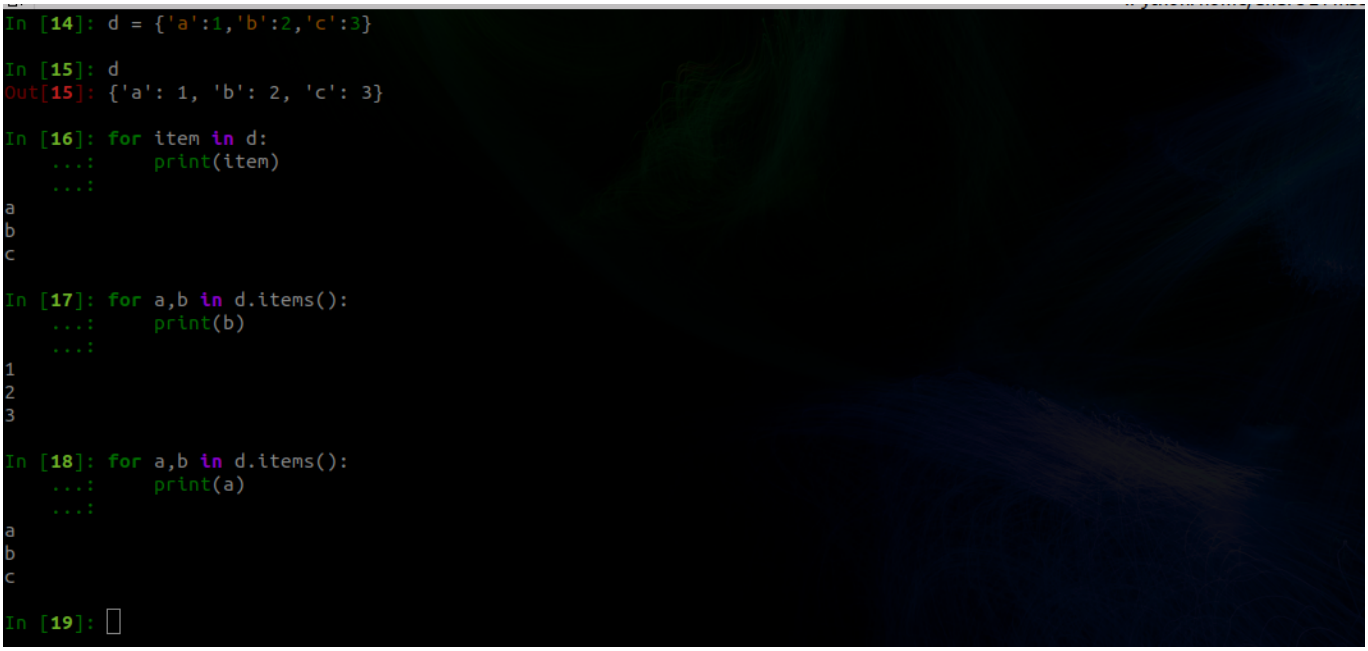

```
mylist = [(1,2,3),(5,6,7),(8,9,10)]
```

```
for a,b,c in mylist:
    print(b)
```

```
2
6
9
```

and example on how to iterate through dictionary

- by default it iterates by keys



- simirarly with dictionayrs if we dont match the structuire with a,b == key:value pairs we will iterate through all of it
- by default if we iterate throught dictionary we get keys
- but when we declare that we have two values a,b in d.items(), note we use function to get the items from dictionary
    - we can manipulate and get ony keys or only values from our dictionary
- this is also tuple unpacking at work
- but remember: dictionarys are unordered so when you iterate on large dictionary there is no guarantee you gonna get results as you put them in

lambdas

The correct way to declare a two-parameter lambda function raising its first parameter to the power of the second parameter is:

D. lambda x, y: x ** y

This lambda function takes two parameters x and y, and returns the result of raising x to the power of y using the ** operator.

Option A is incorrect because lambda is not defined with an assignment operator.

Option B is close but is missing the colon : that separates the argument list from the expression.

Option C is incorrect because it uses the def keyword which is used for defining a regular function, not a lambda function.

Option D is the correct syntax for defining a two-parameter lambda function that raises the first parameter to the power of the second parameter.

lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

One time use function that you quyickly use it and never reference again

Anonymous function:

- we dont give it a name
- we dont start it with def keyword
- simply start with keyword lambda
-

```
x = lambda a : a + 10
print(x(5))
```

lambda example walk through

```
def foo(x,y):
  return (y(x) + y(x+1))

print(foo(1, lambda x: x*x))

The function foo takes two arguments, x and y. x is just a regular
variable, while y is a function that takes one argument.

In the body of the function, y(x) is the first function call. This calls
the function y with x as its argument. In the example code, y is a lambda
function that squares its argument, so y(x) computes the square of x.

The second function call is y(x+1). This calls the same y function, but
with x+1 as its argument. In the example code, this computes the square of
x+1.

Finally, the two results from the function calls are added together, and
the sum is returned by foo.

In the example code, foo(1, lambda x: x*x) is called. This passes 1 as the
value of x and a lambda function that squares its argument as the value of
y. Therefore, y(x) evaluates to 1 squared, which is 1, and y(x+1) evaluates
to 2 squared, which is 4. The sum of these two values is 5, which is the
value returned by foo.

So, the output of the code when run is 5.
```

lambda example walk through 2

```
def foo(x,y):
  return y(x) +(x+1)
print(foo(1,lambda x: x*x)

The function foo takes two arguments, x and y. x is just a regular
variable, while y is a function that takes one argument.

In the body of the function, y(x) is the first function call. This calls
the function y with x as its argument. In the example code, y is a lambda
function that squares its argument, so y(x) computes the square of x.

Then, (x+1) is added to the result of y(x). In the example code, x is 1, so
(x+1) evaluates to 2. Therefore, the final result of foo is the sum of y(x)
and (x+1).

In the example code, foo(1,lambda x: x*x) is called. This passes 1 as the
value of x and a lambda function that squares its argument as the value of
y. Therefore, y(x) evaluates to 1 squared, which is 1. Adding (x+1), which
is 2, we get the result 3, which is the value returned by foo.

So, the output of the code when run is 3.
```

WHILE LOOPS

- while something is true continue
- but we have to remember to always put condition that will end the loop (x = x+1) so the loop ends

```
x = 0

while x < 5:
    print(f'The current value of x is {x}')
    x = x + 1
```

```
The current value of x is 0
The current value of x is 1
The current value of x is 2
The current value of x is 3
The current value of x is 4
```

conditional expression

```
x = 3 % 1
y 1 if x > 0 else 0


# The first line assigns the value of the remainder of the division of 3 by
1 to the variable x. Since 3 is greater than 1, the result of this
calculation is 0.

# This code calculates the remainder of 3 divided by 1, which is 0. The
value of x is then 0. The second line of code uses a ternary operator to
assign either 1 or 0 to the variable y. If x is greater than 0 (which is
not the case here), y will be assigned the value 1. Otherwise, y will be
assigned the value 0. Since x is 0 in this case, y will be assigned the
value 0.
```

BREAK CONTINUE PASS

CONTINUE:

- goes to the top of the closest enclosing loop

BREAK

- breaks out of the current closest enclosing loop

- it stops the loop when leter is equal to a

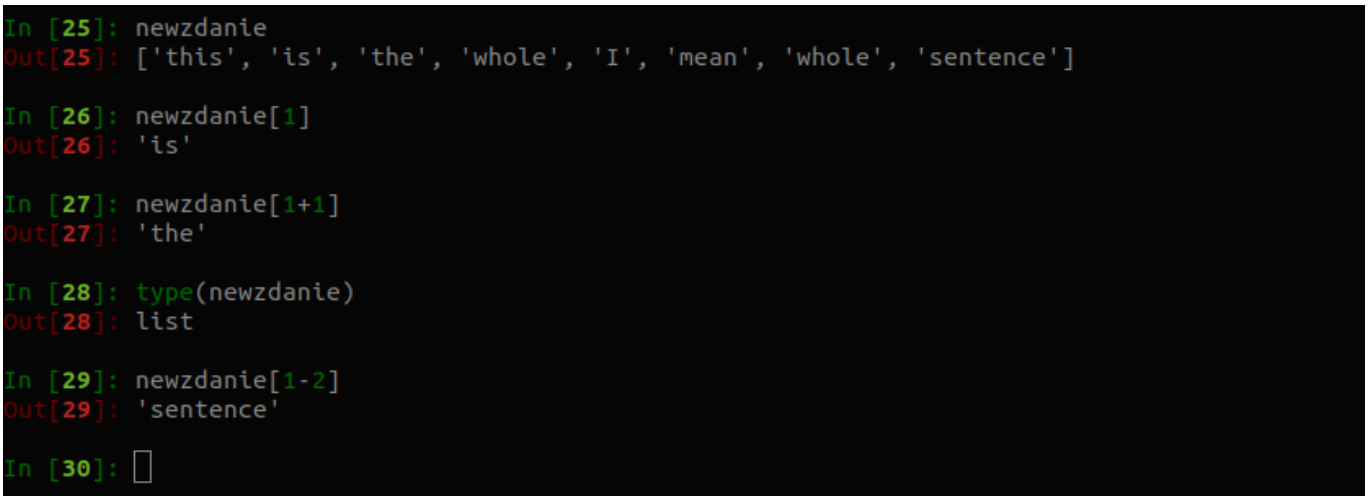
PASS:

- does nothing at all
- good as placeholder for loops or functions that dot work yet but with pass python wont break your code

---

LISTS

- lists are declared with []
- list is a container object for other objects
- list is ordered sequence of elements
- ITEMS IN LIST ARE SAVED IN INDEX POSITION,
- each element has index, 0 based index
- select items from list using that index
- what you can do with index is call object by index
- or call object +1 czyli nie dodajemy jeden tylko nastepny index:




```
#
# create a list from something, anything
#
>>> andre = 'andre'
>>> andre
'andre'
>>> list('andre')
['a', 'n', 'd', 'r', 'e']
>>>


andre = ['handsome', 'dedicated', '5AC']
andre[1]
'dedicated'

# get elements from to EXCLUDING the last one, so always go one more
andre[0:1]
['handsome']

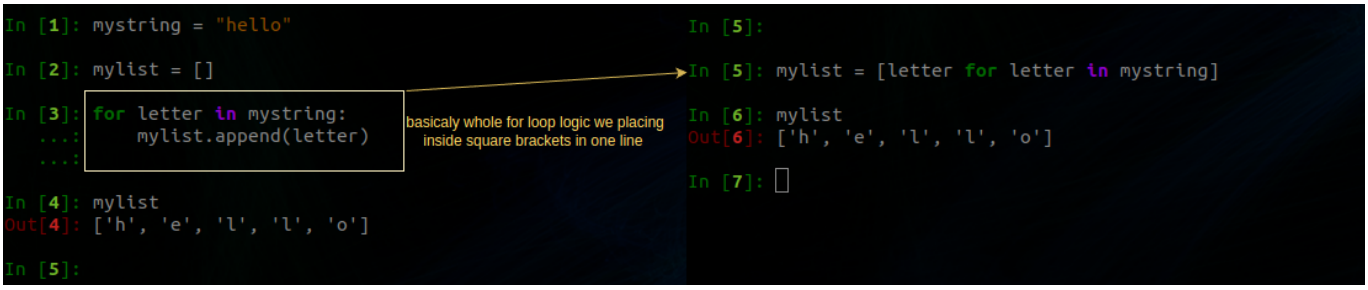
andre[0:2]
['handsome', 'dedicated']

# select last element from list:
andre[-1]
'5AC'

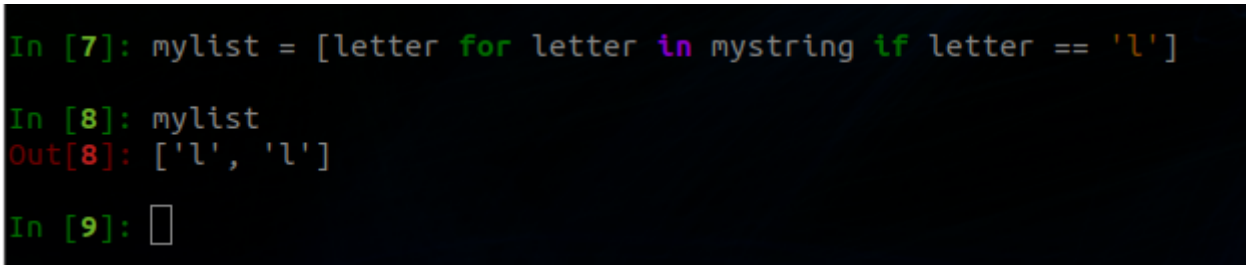
- lists are mutable, meaning we can change them (strings are immutable -
can not be changed)
- so we can change lists inplace, without destroying the object
```python
# change handsome to fit
andre[0] = 'fit'
andre
['fit', 'dedicated', '5AC']
```

LIST COMPREHENSIONS
```

- logic here is essentially flattened out for loop
- element for element in itterable object:



- list comprehension takes this element for elemnt in a list
- if we want to add some more logic



- *if statement* it goes to the right
- letter for letter in my string ONLY IF the letter is ...easy

```
 andre
['fit', 'dedicated', '5AC', 'dedicated', 'handsome', 'fit', 'wolf']
 [a for a in andre]
['fit', 'dedicated', '5AC', 'dedicated', 'handsome', 'fit', 'wolf']


 ['andre is '+a for a in andre]
['andre is fit', 'andre is dedicated', 'andre is 5AC', 'andre is
dedicated', 'andre is handsome', 'andre is fit', 'andre is wolf']
```

TUPLES

- () or nothing
- another type of container (another data structure) that is very similar to lists
- ordered
- immutable, they can not be changed once assigned
- just like lists, tupple has zero based index
- so why touples if we have lists?
    - tuples used when there is always the same data structure that wont be changed
    - but for comparison purposes
- Python, tuples are immutable sequences of elements, which means they cannot be changed after they are created. However, you can perform several operations on tuples, such as:

Accessing Elements: You can access the elements of a tuple using indexing or slicing.

Concatenation: You can concatenate two or more tuples using the "+" operator.

Repetition: You can repeat a tuple multiple times using the "*" operator.

Length: You can find the number of elements in a tuple using the "len()" function.

Membership: You can check if an element is present in a tuple using the "in" keyword.

Sorting: You can sort the elements of a tuple using the "sorted()" function.

Minimum and Maximum: You can find the minimum and maximum elements of a tuple using the "min()" and "max()" functions, respectively.

Counting: You can count the number of occurrences of an element in a tuple using the "count()" method.

Note that since tuples are immutable, you cannot modify, add, or remove elements from a tuple.

```
andre2
('andre', 38, 'whats that')

type(andre2)
<class 'tuple'>

# getting element from tuple, just like from list, use square brackets
andre2[0]
'andre'
```

Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered* and changeable. No duplicate members.

*As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

SETS

- declared with {}
- comma separated elements
- unordered container of only unique values

```
# sets
>>> andre3 =
{"networking","automation","upgrades","performance","networking"}

>>> type(andre3)
<class 'set'>

>>> andre3
{'performance', 'networking', 'automation', 'upgrades'}
#
# networking only once because sets are UNIQUE
#
# getting items by index or position does not work here, because its
unordered
#
>>> andre3[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable

# add / remove from set
andre3.add('APIs')
andre3.discard('APIs')


# when we have two sets we can compare them
#
#  get whats in common with INTERSECTION :
#
>>> andre3
{'performance', 'networking', 'automation', 'upgrades'}
>>> andre4
{'networking', 'automation', 'APIs'}

>>> andre3.intersection(andre4)
{'networking', 'automation'}

# UNION is a bigger set containing all elements from both
#
>>> andre3.union(andre4)
{'performance', 'networking', 'automation', 'upgrades', 'APIs'}

# DIFFERENCE
#
# this goes two ways ALL ELEMENTS THAT ARE IN 1 BUT NOT IN 2
#
>>> andre3.difference(andre4)
{'performance', 'upgrades'}
>>>
>>> andre4.difference(andre3)
{'APIs'}
>>>
```

```
# change list into set is soo easy:
>>> andre
['fit', 'dedicated', '5AC', 'dedicated', 'handsome', 'fit', 'wolf']
>>> type(andre)
<class 'list'>
>>>
>>> set(andre)
{'dedicated', 'fit', 'wolf', '5AC', 'handsome'}
```

DICTIONARY

- {}
- mutable and unordered
- with **Key:Value** pairs
- you can get data from dictionary using Key or .get() method,
- **.get()** doesnt give error so when there is no value to get the program will continue

```
>>> andre11 = {'who':'me', 'where':'here','what':'pandas','why':'to
automate the job'}

>>> andre11['why']
'to automate the job'

>>> andre11.get('why')
'to automate the job'
#
# whether we dont know if the key exists in dictionary we better use .get()
#
>>> andre11['5w']
Traceback (most recent call last):>>> andre11['5w']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: '5w'
>>> andre11.get('5w')

  File "<stdin>", line 1, in <module>
KeyError: '5w'
>>> andre11.get('5w')
#
# adding stuff to dictionary
#
>>> andre11['4W'] = "this is the 4 W's dict"

>>> andre11
{'who': 'me', 'where': 'here', 'what': 'pandas', 'why': 'to automate the
job', '4W': "this is the 4 W's dict"}

# remove from disctionary:
```

```
#
>>> andre11.pop('4W')
"this is the 4 W's dict"

>>> andre11
{'who': 'me', 'where': 'here', 'what': 'pandas', 'why': 'to automate the
job'}
#
# dictionaries can contain any value or container, even disctionary inside
of dictionary
#
andre12 = {
  'who': 'me',
  'where': 'here',
  'what': 'pandas',
  # to jest moje why i tak to wyciagamy: andre12.get('why')
  'why': {
  'why':'to automate the job',
  'morning why':'because I always wanted to code',
  'motiv':'I want to create',
  ('andre', 'me'): {'expertise':'networking', 'good at':'upgrades', 'great
at':'visionary thinking'},
  10: [15, 3],
  }
}

>>> andre12
{'who': 'me', 'where': 'here', 'what': 'pandas', 'why': {'why': 'to
automate the job', 'morning why': 'because I always wanted to code',
'motiv': 'I want to create', ('andre', 'me'): {'expertise': 'networking',
'good at': 'upgrades', 'great at': 'visionary thinking'}, 10: [15, 3]}}
#
# the key can BE IMMUTABLE DATA TYPE - so tuple, or string, or int or even
a dictionaries BUT NOT LISTS
#
>>> andre12
{'who': 'me', 'where': 'here', 'what': 'pandas', 'why': {'why': 'to
automate the job', 'morning why': 'because I always wanted to code',
'motiv': 'I want to create', ('andre', 'me'): {'expertise': 'networking',
'good at': 'upgrades', 'great at': 'visionary thinking'}, 10: [15, 3]}}

>>> andre12.keys()
dict_keys(['who', 'where', 'what', 'why'])

>>> andre12.values()
dict_values(['me', 'here', 'pandas', {'why': 'to automate the job',
'morning why': 'because I always wanted to code', 'motiv': 'I want to
create', ('andre', 'me'): {'expertise': 'networking', 'good at':
'upgrades', 'great at': 'visionary thinking'}, 10: [15, 3]}])

>>> andre12.items()
dict_items([('who', 'me'), ('where', 'here'), ('what', 'pandas'), ('why',
{'why': 'to automate the job', 'morning why': 'because I always wanted to
code', 'motiv': 'I want to create', ('andre', 'me'): {'expertise':
```

```
'networking', 'good at': 'upgrades', 'great at': 'visionary thinking'}, 10:
[15, 3]})])
>>>
#
# getting data from dictionary - ALWAYS PROVIDE THE KEY
#
>>> andre12.get('why')
{'why': 'to automate the job', 'morning why': 'because I always wanted to
code', 'motiv': 'I want to create', ('andre', 'me'):
{'expertiseknownissues_logs = knownissues['logs']': 'networking', 'good
at': 'upgrades', 'great at': 'visionary thinking'}, 10: [15, 3]}
```

TEST STATEMENT

scope determines visibility of that variable name to other parts of the code

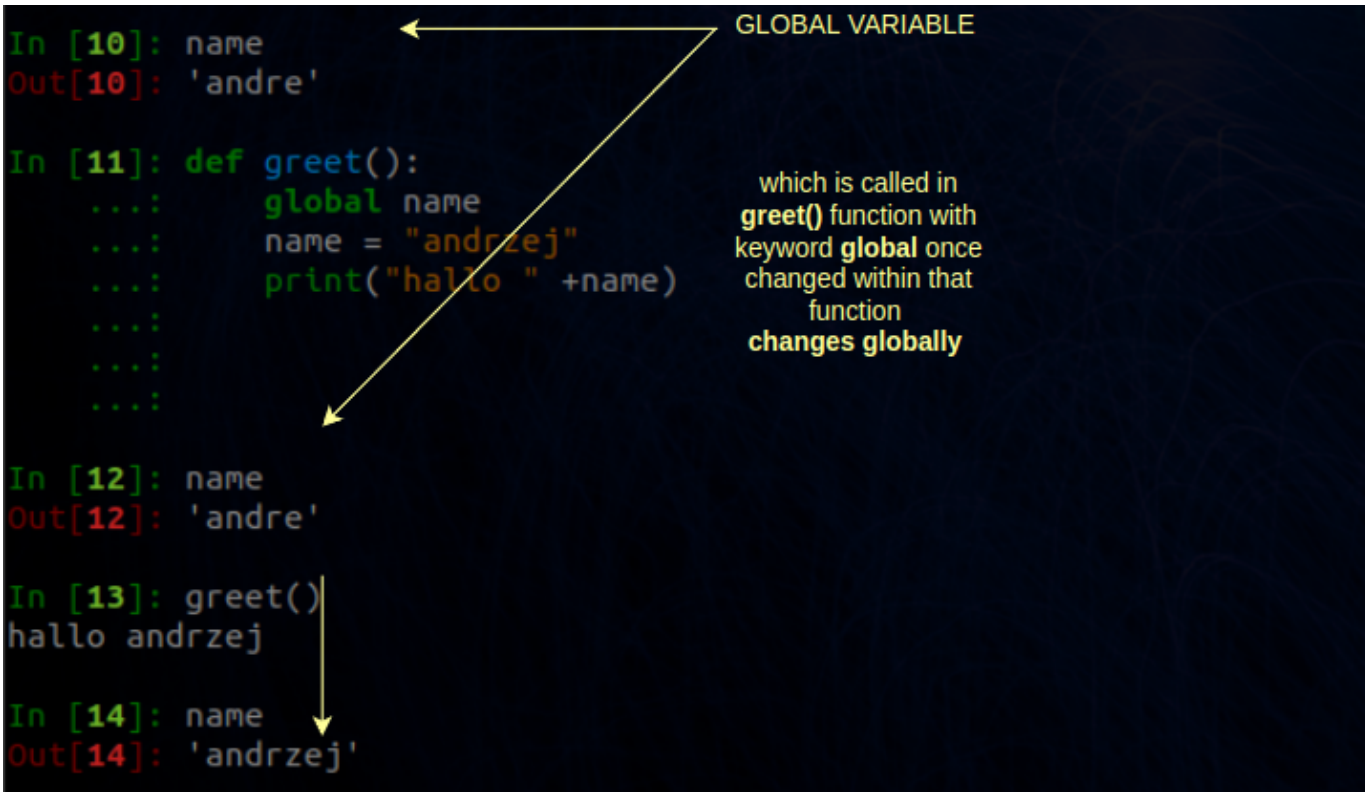
LEGB RULES: >>THE ORDER IN WHICH PYTHON LOOKS FOR VARIABLE NAMES <<

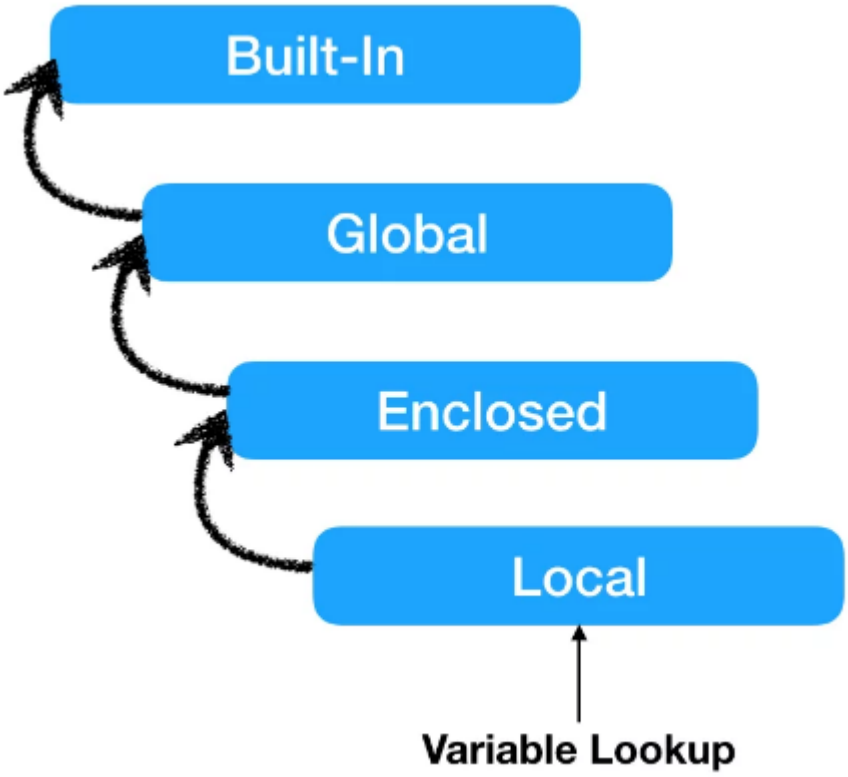
- **L**: local - names assigned in any way within a function and NOT declared global in that function
- **E**: Enclosing - function locals, names in the local scope of any and all functions
- **G**: Global - names assigned at the top-level of module file OR declared global in a def

- **B**: Build-in - all Pyhthon preassigned build-in names, *open, range, sum*

MORE EXPLANATION:

- when you create variable in Python that variable name is stored in *namespace*
- and that variables have a scope, scope determines visibility of that variable name to other parts of the code
- FIRST python looks for the namespace in **local** variable, this names are assigned in any way within a function (def, lambda)
- NEXT in **Enclosed Function locals** variables in function inside the function
- THAN **GLOBAL** declared at the top-level of module file or declared global in def within a file
- and LAST in **BUILD IN** functions (like print(), sum()), any function that you can call help on it is build in `help(sum())`
- local assignments does not affect global ones even when they have the same namings



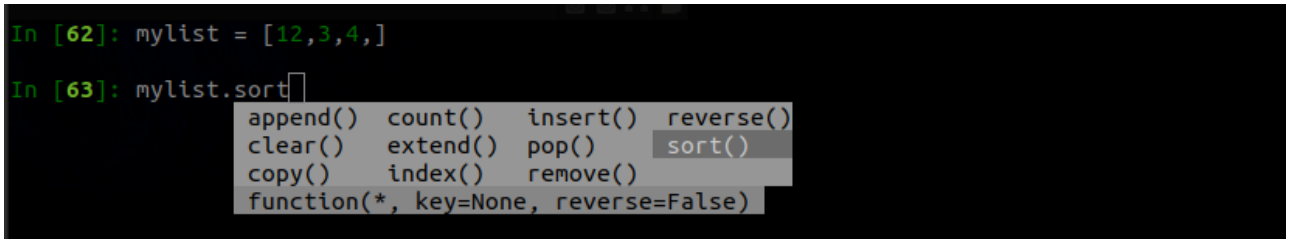


METHODS

METHOD is a function defined inside a class body, if called as an attribute of an instace of that class, method will get the instance object as its first argument, which usualy is (self)

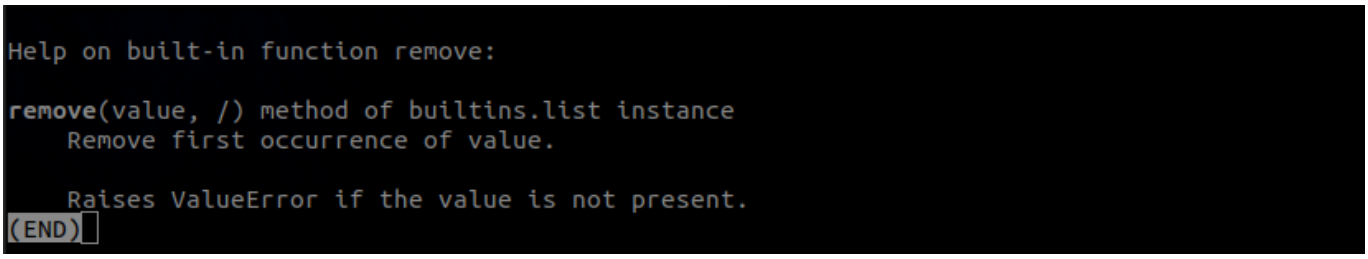
METHODS ARE FUNCTIONS BUILD INTO OBJECTS

- to find build in methods that can be used for each object just type object **.** and hit tab,
- are similar to functions, also have parenesis **()**
- methods are *always attached to object* for example method UPPER attached to object STR
    - `'andre'.upper()`
- you have to provide the object first, then call .method



- and if you dont know how to use the method just call *help()* function:

```
help(mylist.remove)
```



- or for help go to python documentation:
    - https://docs.python.org/3/

The first parameter of each method: A. holds a reference to the currently processed object B. is always set to None C. is set to a unique random value D. is set by the first argument's value

The answer is A. holds a reference to the currently processed object. In Python, the first parameter of an instance method refers to the instance of the class that is calling the method. This parameter is usually called "self" by convention. It is used to access instance variables and other instance methods of the same object.

method vs function

In Python, a function is a block of reusable code that performs a specific task, whereas a method is a function that belongs to an object and is called on that object. In other words, a method is a function that is associated with an object and operates on the data within that object, while a function is a standalone piece of code that can be used anywhere in the program.

Here's an example of a function in Python:

python Copy code def add_numbers(x, y): return x + y This function takes two arguments, x and y, and returns their sum.

And here's an example of a method in Python:

python Copy code my_list = [1, 2, 3] my_list.append(4) Here, my_list is an object of the list class, and the append() method is called on that object to add the value 4 to the end of the list.

In summary, the key differences between methods and functions are:

A function is a standalone block of code that can be called from anywhere in the program, whereas a method is a function that is associated with an object and operates on the data within that object. Methods are called using the dot notation (e.g., my_list.append(4)), whereas functions are called by their name (e.g., add_numbers(2, 3)).

The first parameter of each method in Python is self. This parameter refers to the instance of the class on which the method is called. It is a convention in Python to use the name self for this parameter, although you could technically use any valid variable name.

Here's an example of a simple class with a method that takes self as the first parameter:

python Copy code class MyClass: def my_method(self): print("Hello, world!") In this example, my_method() takes self as its first parameter, although it does not actually use it. When the method is called on an instance of the class, Python automatically passes the instance as the first argument.

Here's an example of how you would call this method on an instance of the class:

python Copy code my_obj = MyClass() my_obj.my_method() # prints "Hello, world!" The second (and any subsequent) parameters of a method depend on the specific method and what it is designed to do. They are defined by the method author and can vary widely depending on the purpose of the method.

---

method examples

- **.format()** method:
  - for substitution example:
    - `"we will be using pythong v{}".format(3.7)` whatever we provide in format method it will be replaced within the brackets {}
    - `"we can do it even better with multiple Key-Word-Arguments, like {this} and {this2} and {this3}".format(this="first key word argument",this2="nother key word argument",this3="can be anything we want")`
- **.append()**

---

arguments in methods

1. Positional arguments:

- Positional arguments are the most common way to pass arguments to a Python function or method.
- When using positional arguments, the values are passed in the order that they are listed in the function or method signature.
- Here's an example:

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

# Call the function using positional arguments
greet("John", 30)

# In this example, the first argument ("John") is passed as the name
parameter, and the second argument (30) is passed as the age parameter.
# The order of the arguments matters - if you swapped the order of the
arguments in the function call, you would get a different resul
```

2. Named arguments:

- Named arguments allow you to specify the parameter name along with the value, so that the order of the arguments doesn't matter.
- Here's an example:

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

# Call the function using named arguments
greet(name="John", age=30)
greet(age=30,name="John")
# the parameter names are explicitly specified in the function call.
# This means that you can pass the arguments in any order, as long as you
use the correct parameter names.
# - both give same result:
Hello, Jophn! You are 30 years old
```

3. Sequential arguments:

- Sequential arguments are used when you want to pass multiple values to a single parameter as a sequence.
- There are two types of sequences commonly used in Python - tuples and lists.
- Here's an example:

```
def greet(names):
    for name in names:
        print(f"Hello, {name}!")

# Call the function using a list of names
greet(["John", "Jane", "Joe"])
# a list of names is passed as a single argument to the greet function. The
function then loops through the list and greets each person individually.
```

4. Keyword arguments:

- Keyword arguments are similar to named arguments, but they allow you to pass a dictionary of parameter names and values instead of individual arguments. Here's an example:

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

# Call the function using keyword arguments
person = {"name": "John", "age": 30}
greet(**person)
#a dictionary containing the parameter names and values is passed to the
greet function using the ** operator. The function then uses the dictionary
keys as the parameter names and the values as the arguments.
```

---

append with MAP() function

```

```

```
andre.append('pythonista')
andre
['fit', 'dedicated', '5AC', 'pythonista']


- **.pop()** removes the item by index
- **.remove()** remove by name
- **.join()** create single string from list
```python
# first thing is the separator
 andre
['fit', 'dedicated', '5AC']
''.join(andre)
'fitdedicated5AC'

'-'.join(andre)
'fit-dedicated-5AC'

'&'.join(andre)
'fit&dedicated&5AC'

' & '.join(andre)
'fit & dedicated & 5AC'
```

```

FUNCTIONS

- **FUNCTION IS A block of organized, resuable code, that runs when you call it**
- function is reusable block of code that performs a specific task.
- Functions allow you to break down complex programs into smaller, more manageable pieces of code
- **Basicaly the main point of functions is to execute block of (reusable) code**
- Functions are object that can not be passed to another object, but you can call function inside another function


```
In [78]: logerror
Out[78]: 'nvme disk is corrupt'

In [79]: def my_function(logerrors):
    ...:     """
    ...:     in this function we are printing in red
    ...:     each logerror we got from logs
    ...:     """
    ...:     print("\nThe error we found in logs is: ", colored(logerror,"red").format(logerror))
    ...:     print()
    ...:     return
    ...:

In [80]: my_function(logerror)

The error we found in logs is:  nvme disk is corrupt

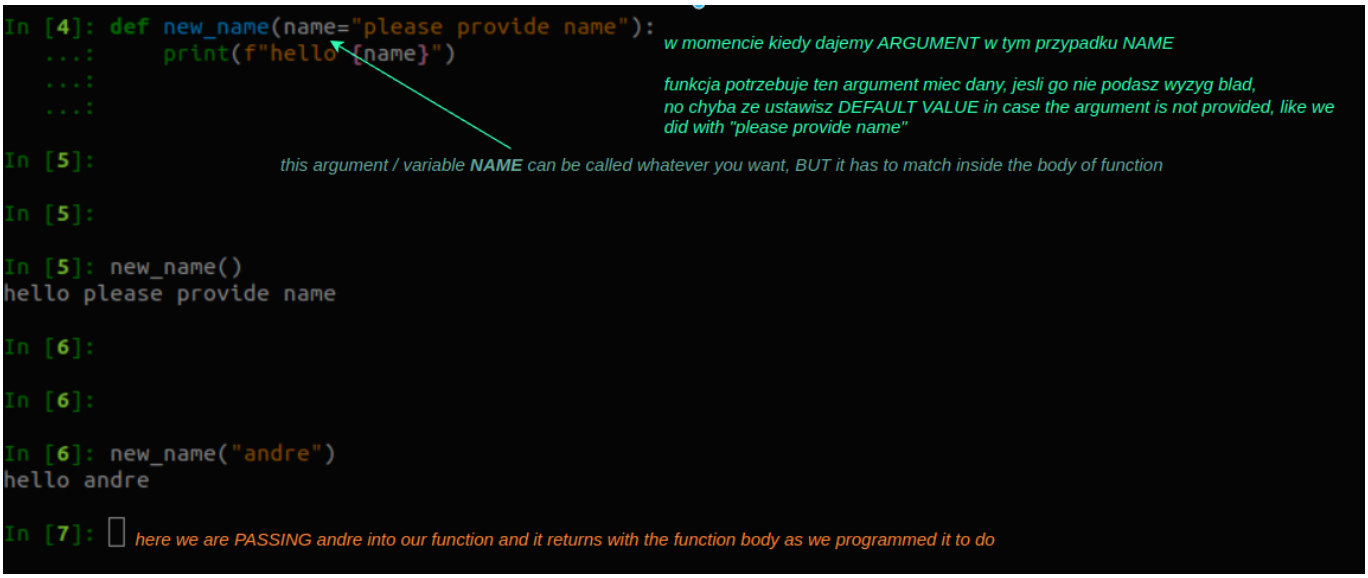

In [81]:
```

```
def f(n):
  if n==1:
    return 1
    return n + f(n-1)
print(f(2))

# The function f takes an integer n as input

# First, the function checks if n is equal to 1 using an if statement. If n
is equal to 1, the function immediately returns 1.

# If n is not equal to 1, the function returns the value of n plus the
result of calling f(n-1) recursively. In other words, the function adds n
to the result of calling itself with n-1 as the argument.
```



THE IMPORTANCE OF **RETURN**

- allows you to save to variable
- using PRINT doesnt allow you to save to variable
- When a return statement is executed the function stops executing and control is returned to the caller.
    - If a value is provided after the return keyword, that value is passed back to the caller as the result of the function.

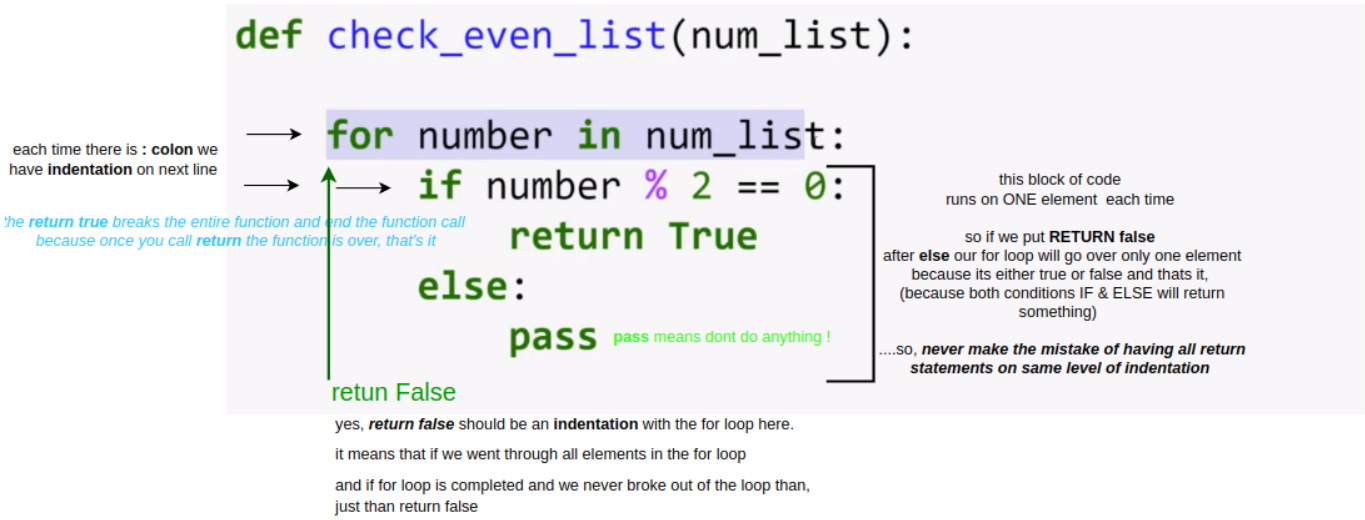
```
def add_numbers(a, b):
    """This function adds two numbers and returns the result."""
    return a + b

# add_numbers() function takes two parameters, a and b, and returns the
result of adding them together using the return keyword.

result = add_numbers(5, 7)
print(result)                  # Output: 12

# This will call the add_numbers() function with the values 5 and 7 as
arguments, and store the result in the result variable. The print()
function is then used to output the value of result to the console.
```

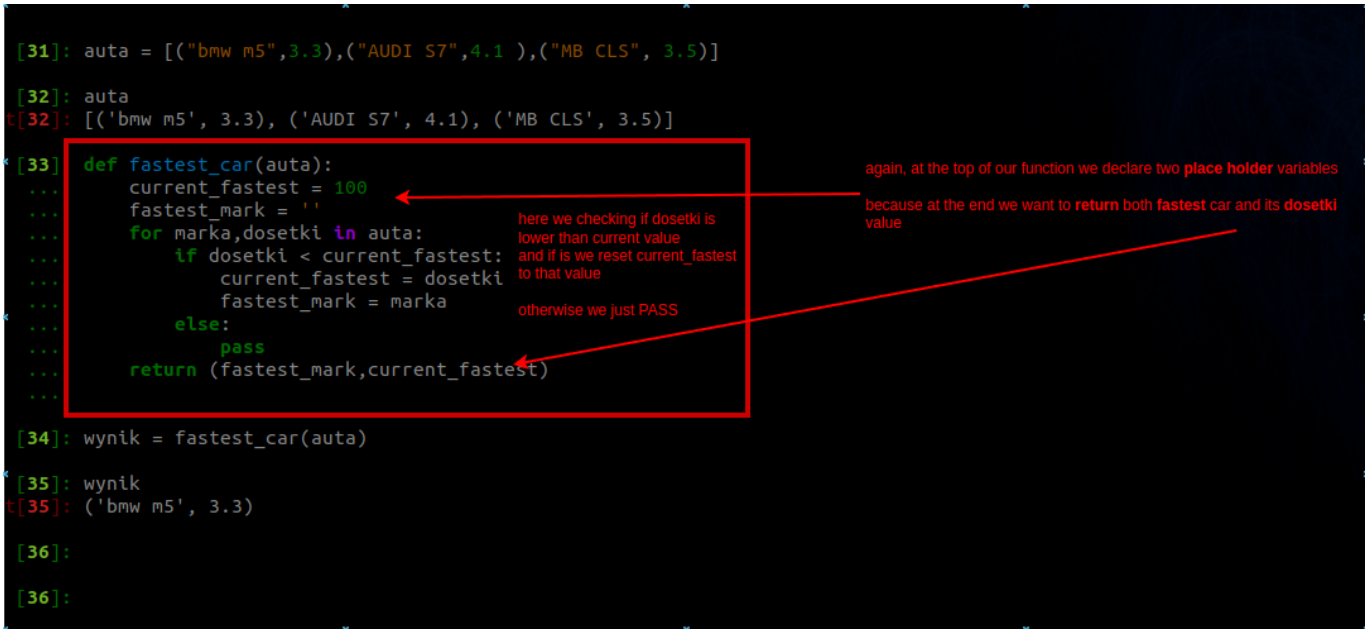
FUNCTIONS LOGIC



KEY POINTS:

- remember about indentations
- if else block
- return has its own indentation
- remember about empty place holder if you want to append anything down the line

RETURNING MULTIPLE ITEMS FROM FUCTION using tuple:



EXAMPLE 4:

```
def foo(x,y):                          # The function foo takes two
arguments, x and y.
     return y(x) + y(x+1)              # y(x) evaluates to 1*1 = 1

print(foo(1,lambda x: x*x))            # y(x+1) evaluates to (1+1)*(1+1) =
```

```
4
#The print statement prints the value 5.

#So, the output of the code is 5. The function foo takes a number x and a
function y, and returns the sum of y(x) and y(x+1). In this case, x is 1
and y is the lambda function lambda x: x*x. So, y(x) is 1*1 = 1 and y(x+1)
is (1+1)*(1+1) = 4, and their sum is 1 + 4 = 5.So, the output of the code
is 5. The function foo takes a number x and a function y, and returns the
sum of y(x) and y(x+1). In this case, x is 1 and y is the lambda function
lambda x: x*x. So, y(x) is 1*1 = 1 and y(x+1) is (1+1)*(1+1) = 4, and their
sum is 1 + 4 = 5.
```

TYPES OF FUNCTIONS:

1. user defined functions
2. Build-in functions
3. Lambda Functions

- hey are anonymous, unnamed

4. recursion functions

- functions that calls themselves
- python has tons of build in functions
- list of build in functions: https://docs.python.org/3/library/functions.html
- most commonly used functions: https://www.positronx.io/useful-python-built-in-functions-and-methods-list/

functions ARGS & KWARGS

ARGS

- pass as many arguments as you want to your function
- example below shows that

```
In [18]: def procenty(a,b):
    ...:
    ...:     piecp = sum((a,b)) * 0.05
    ...:     print("5% of the sum of your numbers is: " +str(piecp))
    ...:
    ...:
    ...:
    ...:
    ...:

In [19]: procenty(1,99)
5% of the sum of your numbers is: 5.0

In [20]: procenty(1,99,100)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-890f37565eff> in <module>
----> 1 procenty(1,99,100)

TypeError: procenty() takes 2 positional arguments but 3 were given

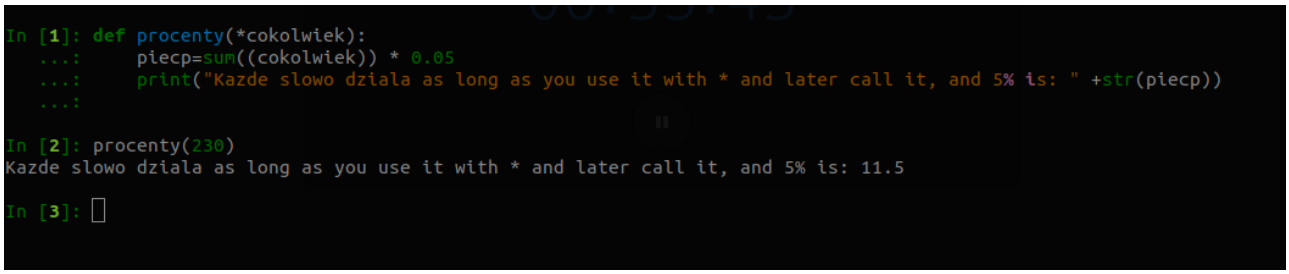
In [21]: def procenty(*args):
    ...:
    ...:     piecp = sum((args)) * 0.05
    ...:     print("5% of the sum of your numbers is: " +str(piecp))
    ...:
    ...:
    ...:
    ...:
    ...:
    ...:

In [22]: procenty(1,99,100)
5% of the sum of your numbers is: 10.0

In [23]: procenty(1,99,100,9000)
5% of the sum of your numbers is: 460.0

In [24]:
```

- **args** can be any word actualy, as long as you have start before it it will work



KWARGS

- same as args just creates dictionary instad of tuples
- we can also mix them toghether,
- **the thing is you have to use them in this order args, kwargs**

```
def myfunc(*args,**kwargs):
    print(args)
    print(kwargs)
    print('I would like {} {}'.format(args[0],kwargs['food']))
```

```
myfunc(10,20,30,fruit='orange',food='eggs',animal='dog')
```

```
(10, 20, 30)
{'fruit': 'orange', 'food': 'eggs', 'animal': 'dog'}
I would like 10 eggs
```

CLASS

- class is a blueprint for creating objects
- It defines a set of attributes and methods that the objects created from the class will have
- An object is an instance of a class, and it can have its own values for the attributes defined in the class.
- The `__init__` method is a special method that gets called when an object of the class is created. It is used to initialize the object's attributes.
- `self` is a reference to the object being created
- `self.name = name` sets name atribute of the object to the value passed in as parameter to the `__init__` method

```
#
# define a Person class with two attributes (name and age) and one method
(say_hello).
#
# __init__ method is a special method that is called when a new object of
the class is created.
# It sets the initial values of the name and age attributes.
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hello(self):
        print("Hello, my name is", self.name, "and I'm", self.age, "years
old.")
# say_hello method is a regular method that takes no arguments (other than
self, which is a reference to the object itself). It simply prints out a
message with the person's name and age.

# To create an object of the Person class, we simply call the class like a
function, passing in the necessary arguments:
p1 = Person("Alice", 30)
p2 = Person("Bob", 25)

# This creates two Person objects, p1 and p2, with different values for the
name and age attributes.

p1.say_hello()  # prints "Hello, my name is Alice and I'm 30 years old."
p2.say_hello()  # prints "Hello, my name is Bob and I'm 25 years old."
```

class hierarchy

```
#
# hierarchy of classes:
#
class A:
    pass
# A is a base class that does not have any defined methods or attributes

class B(A):
    pass
# B is a subclass of A.

class C(A):
    pass
# C is another subclass of A.

class D(B,C):
    pass
# D is a subclass of both B and C, which means it inherits from both of
them.

class Class_3(A,C):
    def __init__(self):
        super().__init__()
# Class_3 is a subclass of both A and C. It has an __init__ method that
calls the __init__ method of its superclasses using super().__init__().

class Class_4(C,B):
    pass
# Class_4 is a subclass of both C and B.

class Class_1(D):
    pass
# Class_1 is a subclass of D.

class Class_2(A,B):
    pass
# Class_2 is a subclass of both A and B.
```

class instance

```
class x:
pass
class y(x):
pass
class z(y):
pass
# Those lines of code define three Python classes x, y, and z.
# The x class has no attributes or methods, and the y and z classes inherit
from x.

# Now lets create two objects, one of type z and one of type x.
x=z()
z=z()
# The first line creates an object of type z and assigns it to the variable
x.
# The second line creates another object of type z and assigns it to the
variable z

isinstance(x,z) isinstance(z,x)
# The first isinstance() call checks whether the object x is an instance of
the z class. Since x was created as an instance of the z class, this call
will return True.

# The second isinstance() call checks whether the object z is an instance
of the x class. However, the object z was created as an instance of the z
class, not the x class. Therefore, this call will return False.

# isinstance(x, z) # True
# isinstance(z, z) # False
```

- remember this rule that whenever there are more than 2 classes from which a class is inheriting from and both classes contain a function/variable with same name then the class name on the left is given priority and its functions/variables are called

```
class A:
    def func(self):
        return "A"

class B:
    def func(self):
        return "B"

class C(B,A):
    pass
```

```
c = C().func()
print(c)

out: B
```

__dict __ in class

In Python, every object (including classes and instances) has a `__dict__` attribute that stores the object's attributes as a dictionary.

This means that all object attributes (both class and instance) can be accessed and modified using the dictionary-like syntax

From the exmple above we can display all atributes

```
Person.__dict__

mappingproxy({'__module__': '__main__',
              '__init__': <function __main__.Person.__init__(self, name,
age)>,
              'say_hello': <function __main__.Person.say_hello(self)>,
              '__dict__': <attribute '__dict__' of 'Person' objects>,
              '__weakref__': <attribute '__weakref__' of 'Person' objects>,
              '__doc__': None})
```

subclasses # superclasses

- **subclass** is a class that inherits properties and methods from a superclass. The subclass can add new properties or methods or override the inherited ones.
- **superclass** is a class that is being inherited from.

For example:

```
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def speak(self):
        print("I am an animal")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name, species="Dog")
        self.breed = breed

    def speak(self):
        print("Woof!")

dog1 = Dog("Fido", "Labrador")
print(dog1.name)      # Fido
print(dog1.species)   # Dog
print(dog1.breed)     # Labrador
dog1.speak()          # Woof!

# Superclass: Animal that has an __init__ method to initialize the name and
species attributes and a speak method that prints "I am an animal".

#  The subclass Dog inherits from the Animal class using the syntax class
Dog(Animal):.

# It has its own __init__ method that calls the super().__init__ method to
initialize the name attribute and set the species attribute to "Dog". It
also has a breed attribute. The speak method is overridden to print
"Woof!".

# We then create an instance of Dog called dog1 with the name "Fido" and
the breed "Labrador". We can access its attributes using the dot notation
and call its methods. When we call dog1.speak(), it prints "Woof!" because
the speak method was overridden in the Dog subclass.
```

isinstance

- instance is individual object from certaing class,
- if a class is a sublcass of another class than an instance of the sublcass is considered an instance of the superclass as well
- so if we have two classes A & B and B is subclass of A object created from B will be instance of A as well:

```
class A:
     VarA = 1
     def __init__(self) -> None:
         self.prop_a=1

class B(A):
     VarA = 2
     def __init__(self) -> None:
         self.prop_b=2

obj_a = A()
obj_aa = A()
obj_b=B()
obj_bb=B()

print(isinstance(obj_b,A))
print(A.VarA==1)
print(obj_a is obj_aa)

True
True
False
```

- but if we change to not be subclass of A, than isinstance(obj_b,A) will return false:

```
#
# but if we change to not be subclass of A, than isinstance(obj_b,A) will
return false:
#
class A:
     VarA = 1
     def __init__(self) -> None:
         self.prop_a=1

class B():
     VarA = 2
     def __init__(self) -> None:
         self.prop_b=2

obj_a = A()
obj_aa = A()
obj_b=B()
obj_bb=B()

print(isinstance(obj_b,A))
print(A.VarA==1)
print(obj_a is obj_aa)
...:
False
True
False
```

```
#
# To check if object is an instance of the Lower class would be to use
isinstance(object, Lower):
#
class Upper:
    def __init__(self):
        self.property = "upper"

class Lower(Upper):
    def __init__(self):
        super().__init__()

object = Lower()
isinstance(object, Lower)     # Returns True
object.property               # Returns "upper"
#
# The two classes: Upper and Lower. Lower is a subclass of Upper, which
means it inherits all of its attributes and methods.
#
# When we create an instance of Lower and assign it to the variable object,
we can check if it is an instance of the Lower class using the isinstance
function with isinstance(object, Lower), which returns True.
#
# Even though object.property returns "upper", the fact that Lower is a
subclass of Upper means that it also has the attribute property with the
same value inherited from Upper. So object.property returns the value that
was set in the __init__ method of Upper.
super().__init__() is a call to the constructor of the superclass of Lower,
which is Upper. In other words, it calls the __init__() method of the Upper
class and initializes the property attribute to the value "upper".

When you create an instance of the Lower class, its __init__() method is
called, and this method then calls the __init__() method of its superclass
(Upper) using the super() function, which returns a temporary object of the
superclass that allows you to call its methods.

So super().__init__() simply means to call the constructor of the
superclass and pass any necessary arguments. In this case, the Upper class
does not take any arguments, so we don't need to pass anything to
super().__init__().
```

CLASS CONSTRUCTORS:

A constructor is a special method that is called when an object of a class is created. In Python, the constructor method is named **init**. The constructor method always takes self as its first parameter. self refers to the instance of the class being created. The constructor can take additional parameters, which can be used to set the initial state of the instance. The **init** method is called automatically when an object of

the class is created using the class name followed by parentheses, like this: my_object = MyClass(). The **init** method can also be used to perform any initialization that is required for the class. If a class does not have a constructor defined, Python will use a default constructor that does nothing. A class can have more than one constructor defined by using class methods as alternative constructors. These methods are often called "factory" methods. Constructors can also be used to perform validation or raise exceptions if invalid arguments are passed to the constructor. Constructors can be overridden in subclass to add additional functionality or override parent class' behavior.

class constructor is a special method that is used to initialize the object's attributes when the object is created. The constructor method is called automatically when an object is instantiated, and it can take parameters to set the initial state of the object.

- The constructor is a special method in Python classes that is used to initialize instance variables when objects are created.
- It is always named `__init__()` and takes self as its first argument.
- The constructor method can take parameters that are used to set the initial state of the object.
- By convention, it should not return anything other than None, If it does, a TypeError will be raised.
- first parameter of a constructor in Python must always be named **self**.

Constructor is a special method that is used to initialize objects when they are created. The constructor method is always named **init**() and it takes at least one argument, self, which refers to the instance of the class that is being created.

Here's an example of a simple Python class that has a constructor:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

In this example, the Person class has a constructor that takes two arguments, name and age. The constructor initializes two instance variables, self.name and self.age, with the values of the name and age arguments.

When you create an instance of the Person class, you pass in values for the name and age arguments:
`person1 = Person("Alice", 25)`

This creates a new instance of the Person class, with self referring to the new instance. The constructor initializes the name instance variable to "Alice" and the age instance variable to 25.

You can access these instance variables using dot notation:

```
print(person1.name)
# Output: Alice

print(person1.age)
# Output: 25
```

class variable vs instance variable

A **class variable** is a variable that is defined within a class but outside of any methods, and its value is shared by all instances of the class. Therefore, you can access it using the Class name, as in Class.variable.

On the other hand, an **instance variable** is a variable that is defined within an instance method or the `__init__` method, and its value is unique to each instance of the class. Therefore, you need to create an instance of the class first, as in instance = Class(), and then access the instance variable using the dot notation, as in `instance.value`.

```
class Class:
  variable = 0            # class variable
  def __init__(self):
    self.value = 0        # instance variable

object1 = Class()         # object1's value is still 0 because it wasnt
changed.
Class.variable += 1       # Class's variable is now 1 because we
incremented it using Class.variable += 1.
object2 = Class()
object2.value += 1        # object2's value is now 1 because we incremented
it using object2.value += 1.

# First, we define a class called Class that has a class variable variable
initialized to 0, and an __init__ method that initializes an instance
variable value to 0.

# Next, we create an object of Class called object1 using the Class()
constructor. This initializes object1 with a value of 0 for value, since
that's what's specified in the __init__ method.

# Then, we increment the class variable variable by 1 by using the syntax
Class.variable += 1. This changes the value of the variable attribute of
the Class class itself, not the variable attribute of any instances of the
class.

# After that, we create a second object of Class called object2. This
initializes object2 with a value of 0 for value, since that's what's
specified in the __init__ method.

# Finally, we increment the value attribute of object2 by 1 by using the
syntax object2.value += 1. This changes the value of the value attribute of
object2, not the value attribute of any other instance of the Class class.
```


In [31]: class Class:
...: variable = 0
...: def __init__(self):
...: self.value = 0
...:
In [32]: object1 = Class()
In [33]: object1.variable
Out[33]: 0
variable value has not changed in object untill we change it << here
In [34]: Class.variable += 1
In [35]: object1.variable
Out[35]: 1
since than all objectX.variable is now 1
In [36]: object2 = Class()
In [37]: object2.variable
Out[37]: 1
In [38]: object2.value +=1
In [39]: object2.variable
Out[39]: 1
In [40]: object1.value
Out[40]: 0
but value is always 0 unless we change that to the specific instance
In [41]: object2.value
Out[41]: 1
In [42]: object3 = Class()
In [43]: object3.value
Out[43]: 0
In [44]: object3.variable
Out[44]: 1

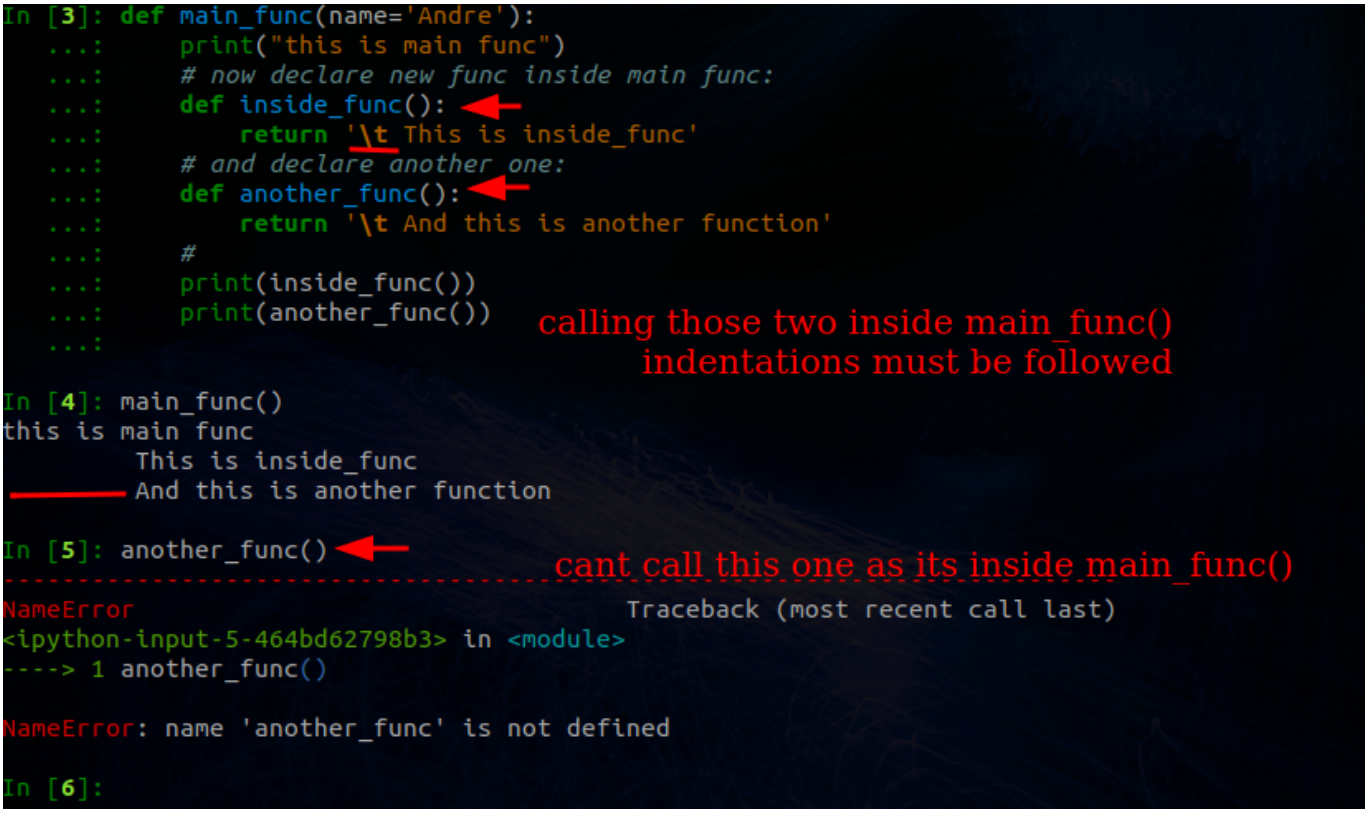
DECORATORS

```
@some_decorator
def simple_func():
    # Do simple stuff
    return something
```

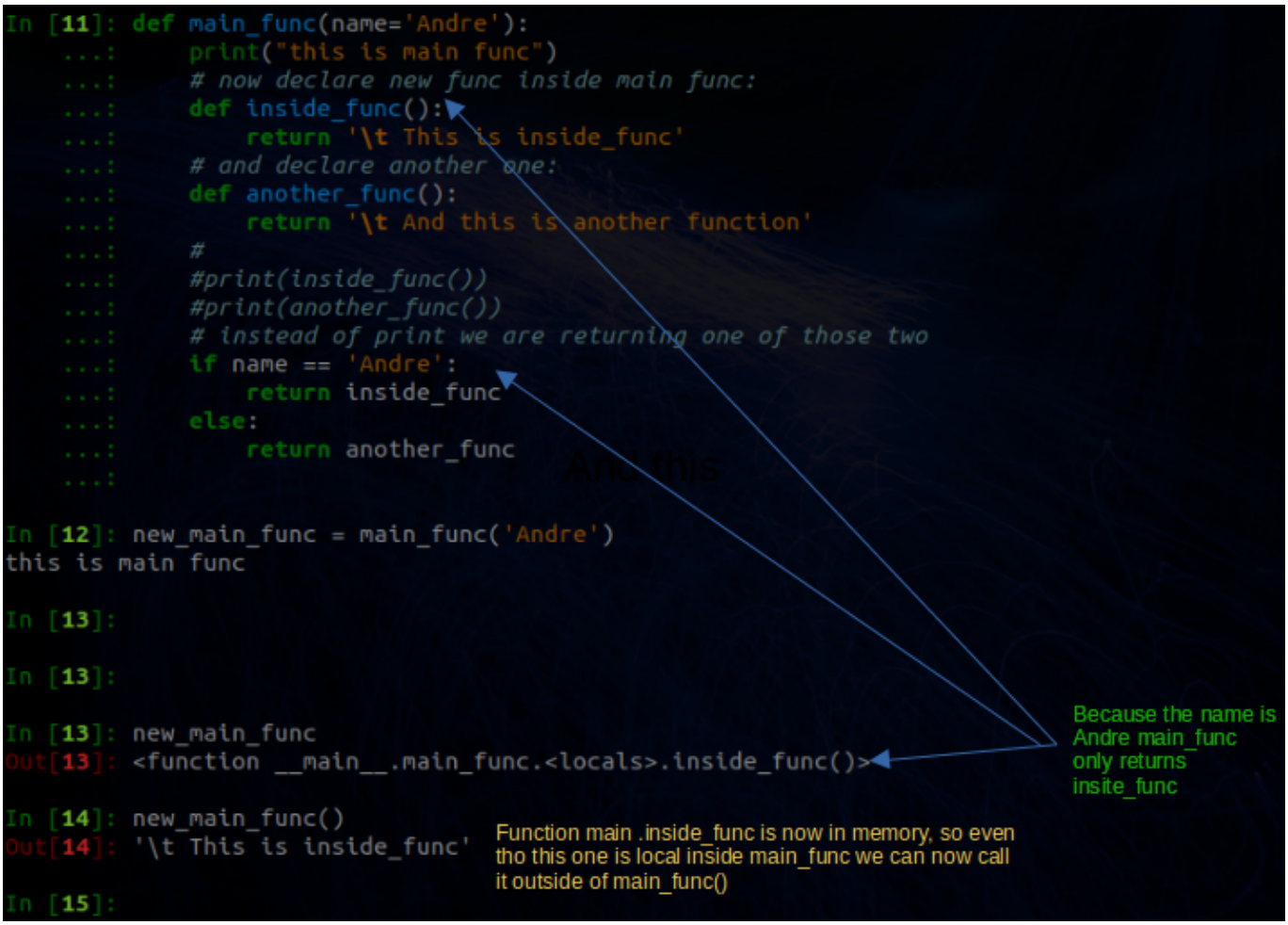
- DECORATORS ALLOW YOU TO DECOTRATE THE FUNCTION
  - decorators allows you to add extra functionality to any existing function
  - decorators are used with **@** sign and are placed on top of original function
  - and than you can delete just one line from decorator if you dont need that extra functionality any more

BEFORE WE GET INTO DECORATORS, LETS SEE HOW WE CAN USE / CALL FUNCTION WITHIN FUNCTIONS:

FIRST WE RETURN FUNCTION:

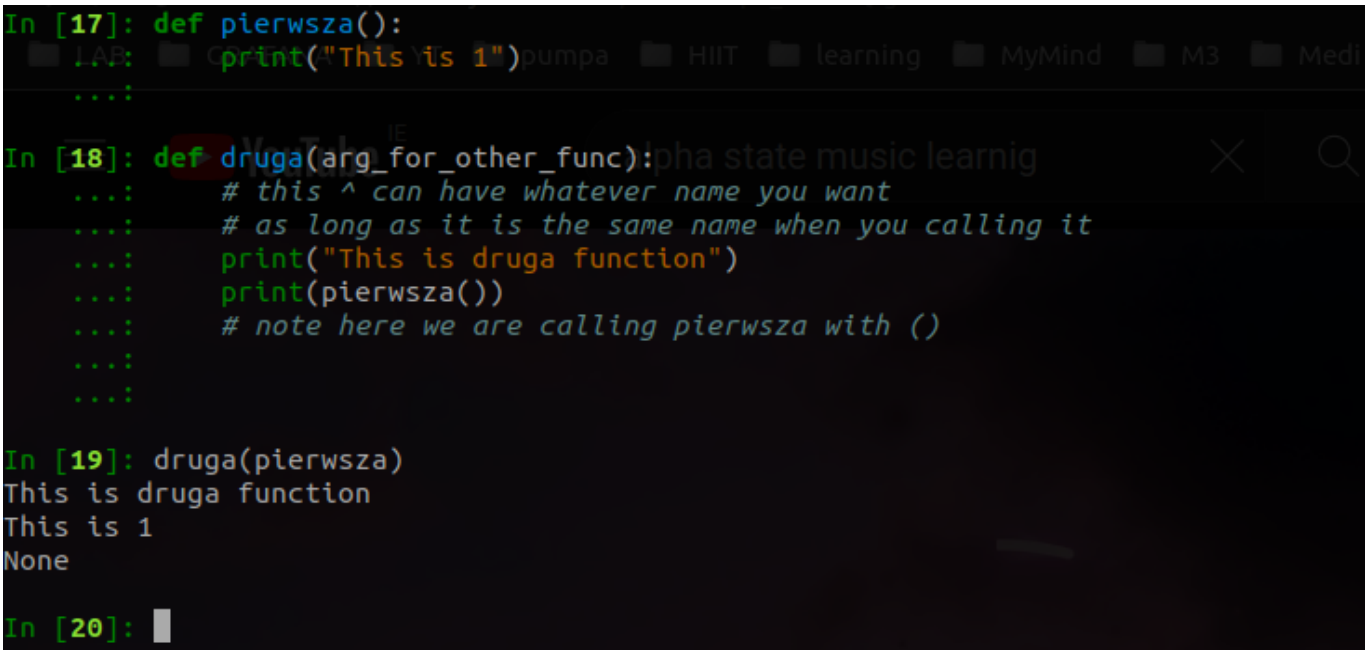


- functions must return something
- functions must be called within
- calling them outside of their indentention wont work

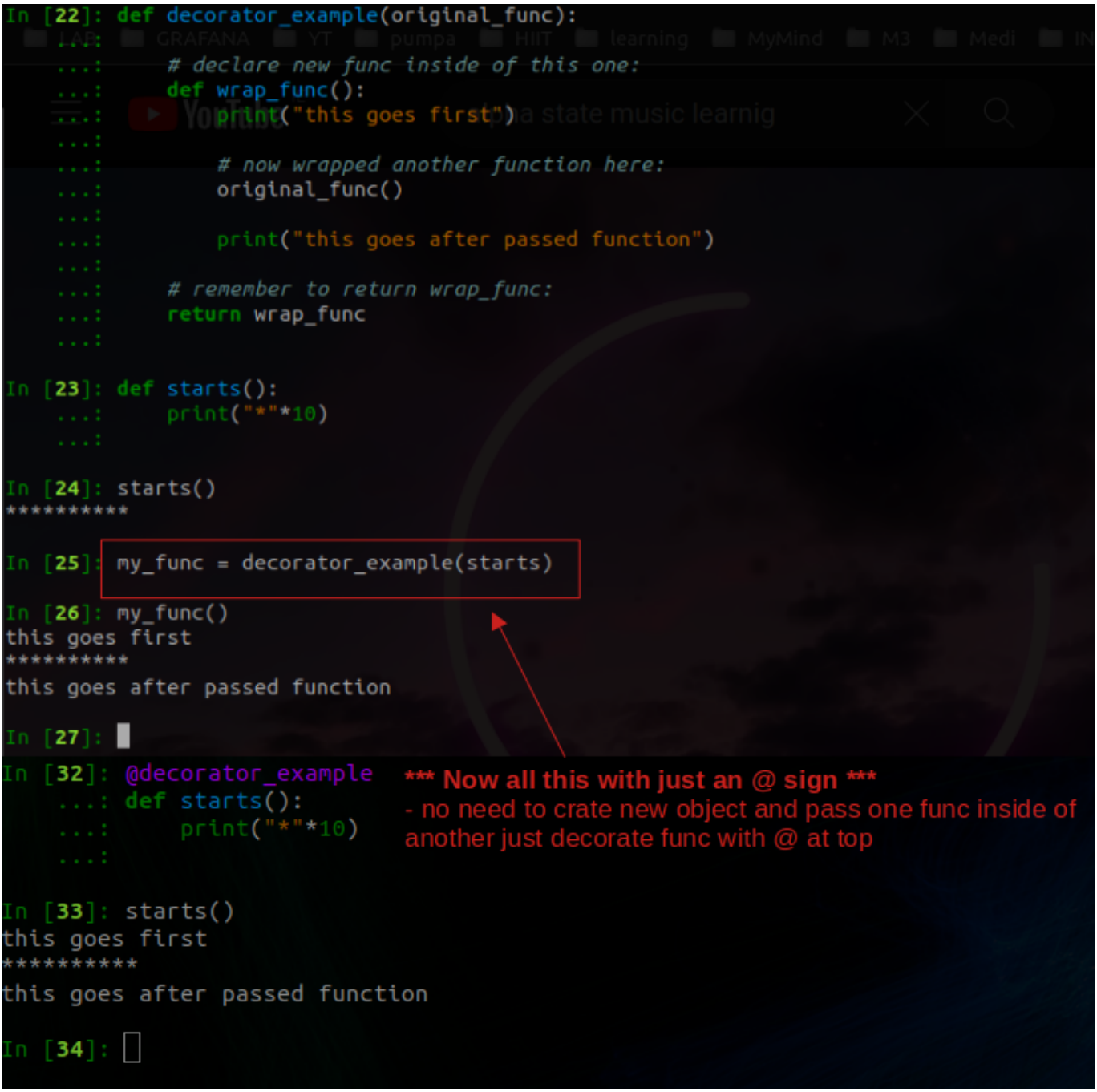


- a little bit more logic here, that if/else returns one or other func
- once func returned and assigned to new object it can be called

NOW, EXAMPLE OF PASSING FUNCTION AS ARGUMENT:



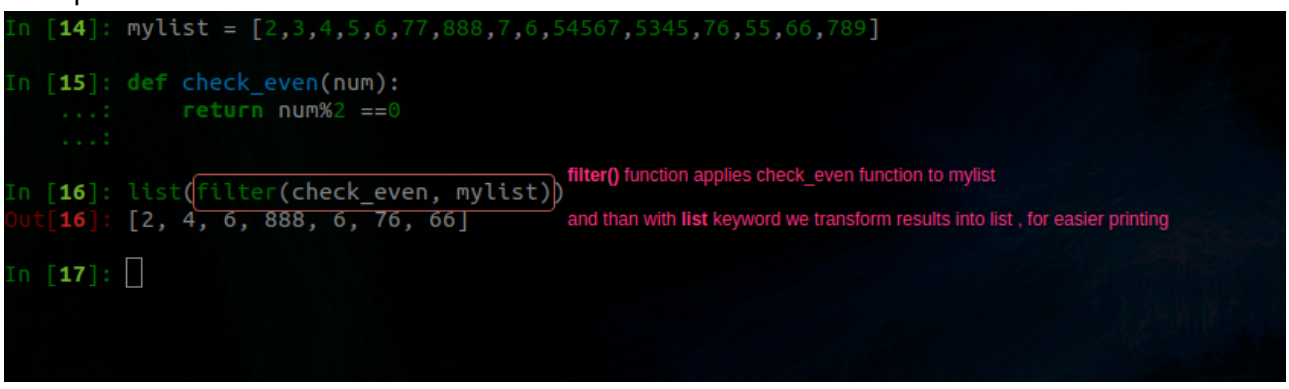
NOW, THE DECORATOR PART:



---

filter()

- filter() function
- filter elements based on condistions of function you pass to it
- example:



IMAGES

- zacznij od `pip3 install pillow` library
- PIL python image library hence pillow
- https://pillow.readthedocs.io/en/stable/ <-- RTFM
- not displayed in ipython 😦

PDF & SPREEDSHEETS

- python can read PDFs and spreadsheets
- CSV comma separated output
- you can use google spreadsheets with python api
- not all PDFs can be readable with python because many of PDFs are encapsulated and displayed in fixed-layout flat document
- also PDFs has different fonts, images, tables etc which is all hard to read for python
- for PDFs use `PyPDF2` library

```
#
# PDF
#

# open pdf as read binary
f = open('Coffee_Break_Python.pdf', 'rb')

# now craete new var and call the pypdf2 library for the file
pdf_reader = PyPDF2.PdfReader(f)

# check how many pages:
len(pdf_reader.pages)
89

# if you want to read pages or search through pages you need to pass them
from pypdf2 into python
get_page = pdf_reader.pages[10]           # provide page number
get_page_text = get_page.extract_text()   # extract the text
get_page_text                             # read the text -- if empty,
means that the extraction didnt really work...
f.close()
#
#
#
# READ ALL TEXT FROM THE PDF FILE WITH LOOP:
f = open('Coffee_Break_Python.pdf', 'rb')

# place holder for the text:
pdf_text = []
```

```
# create reader object:
pdf_reader = PyPDF2.PdfReader(f)
# for loop from range of all pages:
for num in range(len(pdf_reader.pages)):
  page = pdf_reader.pages[num]
  pdf_text.append(page.extract_text())

# now we have it all in
pdf_text
# and we can call by pages from pdf_text with index
pdf_text[11]
# and to make it more readable just print it
print(pdf_text[11])
```

CSV files

```
import csv
dta = open('location_of_my_csv',encoding='utf-8')
csv_data = csv.reader(data)
data_lines = list(csv_data)
```

files readline()

readline() method returns ONE line from the file

if you specify byte size, it will read bytes from flile

```
# having abc.txt file:
abc
def
ghi
# and reading the file with readline(3) will read three bytes
file = open('abc.txt')
print(file.readline(3))
out: abc
```