Introduction to Python

Part 1

Introduction to Python

- What is Python?

  "Python is an interpreted, object-oriented, high-level programming language with dynamic semantics." - python.org

- What can Python be used for?
  - Almost everything! Major applications are…
    - Scripting
    - Web development
    - Scientific and numeric computing
    - Teaching programming
    - Data science, AI and Machine Learning
    - Game development, desktop applications, embedded systems and etc.

Introduction to Python

- An interpreted scripting language.
  - Code is translated by an interpreter to byte-code and then executed by the interpreter
    - Code is written and then directly executed by an interpreter
    - Type commands into interpreter and see immediate results
- An open source, portable, object oriented and functional programming language
- Compilation to portable byte-code is possible
  - To increase the execution speed and protect source codes
- Interactive interface to Python with a REPL (Read–eval–print loop)

- Running Interactively
- Running Scripts (files)

```
iustm@Mehri MINGW64 ~/OneDrive/Desktop
$ python
Python 3.9.4 (tags/v3.9.4:1f2e308, Apr
Type "help", "copyright", "credits" or
>>> x = 2
>>> y = 3
>>> print("x + y = ", x + y)
x + y =  5
>>>
```

example.py

```
1   #! path to python. e.g. /usr/bin/python
2
3   x = 2
4   y = 3
5   print("x + y =", x + y)
```

```
iustm@Mehri MINGW64
$ python example.py
x + y = 5

iustm@Mehri MINGW64
$ ./example.py
x + y = 5
```

Introduction to Python

- **pip** is the package manager for Python packages. Look at here
- Possible to create executable file from a Python script using **pyinstaller** package
  - To install the package run: ***pip install pyinstaller***
  - To make an executable file from a script run: ***pyinstaller --onefile the_script.py***
- Possible to compile a python script to byte-code using **py_compile** and **compileall**
  - The interpreter generates **.pyc** files (python byte-code)
  - In code or directly using the interactive interpreter of python:
    - Using py_compile: **py_compile.compile('abc.py')** or **py_compile.main(['File1.py','File2.py','File3.py'])**
    - Using compileall: **compileall.compile_dir(directoryname)**
  - In a terminal: **python -m py_compile File1.py File2.py …** or **python -m compileall**
    - **compileall** automatically goes recursively into sub directories and make .pyc files for all py files

Introduction to Python

- Python 2 and Python 3
  - Python 2.7 first released in 2010, last and final update 2019-10-19
  - Python 3.0 first released in 2008
  - Python 3.9 first released in 2020, last update 2021-04-04 (Python 3.9.4)
  - Python 2 is now deprecated since end of 2019 but some legacy code still exists
  - Only use Python 3 and upgrade anything Python 2 to Python 3
  - You will learn Python 3 in this course

- What you will learn …
  - Basic operators, Variables and Types, String and string formatting, User Input and List
  - Tuples, Dictionaries, Sets, Conditions, Loops, Exit Codes, Exception Handling and Functions
  - File I/O, Packages, Modules, List Comprehension and Multiple Function Arguments

Introduction to Python

- Python does not have a main function like C
  - The program's main code is just written directly in the file
- Python statements do not end with semicolons
- The print function is used to display text to the output
  - print(var1, var2, ...) - print() # an empty line
  - print("Hello, world!") or print('Hello, world!')
  - Backslashes can be used to escape " and ' (\" or \')
    - E.g. print("Python is a \"scripting\" language.")
- Comments: #, """ or ''' are used for commenting
- Python uses **indentation** to indicate blocks, instead of {}
  - Use TAB for indentation.

```
# x is 2
x = 2
y = 3  # y is 3

'''
print sum of
x and y
'''

"""
print sum of
x and y
"""

print("x + y =", x + y)
```

Introduction to Python

- Basic data types
  - Numbers
    - Integers (default for numbers). E.g. x = 2
      - Python uses arbitrary precision integers. E.g. print(42 ** 999)
    - Floats. E.g. x = 3.1415
      - Floats are not arbitrary precision. IEEE 754 is used (53 bits of precision)
      - E.g. print(4.2 ** 999) => OverflowError: (34, 'Result too large')
    - Complex. E.g. *print(1 + 2j)* or *z = complex(x,y)*
  - Strings: Text enclosed in "" or ''
    - Unmatched can occur within the string. E.g. print("It's a function")
  - Boolean: True, False – boolean representation of true or false
  - None: A special type representing nothing or empty (Used to define a null, or no value)

```
x = 2
y = 3.1415
z = complex(x, y)
s = "Hello World!"
b = True
n = None
```

Introduction to Python

- Variables
  - Python uses dynamic typing, a variable can be reassigned to another type.
  - Variable names are case sensitive, unique and cannot start with a number
  - Variable names can contain letters, numbers, and underscores.
  - A variable name can not be a reserved keyword. E.g. if, while, for, import and etc.
- Naming conventions
  - Variables and functions use lowercase with underscore (e.g. my_name)
  - Constants are capitalized, e.g. MAX_TEMPERATURE
- To get type of a variable you can use **type**(). E.g. type(m_name) => str
- Casting between integers, floats and strings - int, float and str
  - E.g. x = int(3.2), x = int("2"), x = float("2.1"), x = str(3.14). print(type(x)) => str

Introduction to Python - Operators

- Arithmetic and bitwise operators

+	Addition	*	Multiplication	**%**	Modulus	**//**	integer division	**|**	Bitwise OR
-	Subtraction	/	Division	**	Exponentiation	**&**	Bitwise AND	**^**	Bitwise XOR
~	Bitwise NOT	**<<**	Shift shift	**>>**	Right shift				

- Assignment and compound assignment operators
  - =, +=, -=, *=, /=, %=, **=, //=, &=, |=, ^=, ~=, <<=, >>=
  - E.g. x = 4, x, y = 2, 3 => x = 2 and y = 3 or x += 2 => x = x +2
- Logical Operators: and, or, not
- Comparison operators: ==, !=, >, <, >=, <=
- Identity operators: **is** and **is not**. E.g. x is True, x is not True, x is y and etc.
- Membership operators: **in** and **not in**. 5 in list:

Introduction to Python - Strings

- We have already seen string constants, e.g. "Hello world!" or 'Hello world!'
- We can create multiline string using triple quotes or lines in parentheses
- To format strings use the format function.
  - print("Num: {}, Text: {}".format(8, "Hello")) => Num:8, Text: Hello
  - { and } are escaped by doubling them: {{ and }}
  - Inside the braces we can have index of variable and add formatting style e.g.
    - print("z = {0:.03f} + {1:.03f} - {2}, {3} and {2}".format(4/3, 5/3, 'A', 'B'))
      - Output => z = 1.333 + 1.667 - A, B and A
  - For more info look at the python documentation
- Strings can be added. E.g. x = "Hello ", y = "world!", print(x + y) => Hello World!
- Strings can be multiplied. E.g. x = "This can be", y = "repeated ", print(x + " " + y * 3)

```
string = '''Hello World!
Hello World!
Hello World!'''
```

```
string = ("Hello World!\n"
          "Hello World!\n"
          "Hello World!")
```

Introduction to Python - Strings

- Concatenate with + or neighbors. E.g. x = me, word = 'Help' + x or word = 'Help' 'me'
- subscripting of strings
  - 'Hello'[2] => 'l'
  - slice: 'Hello'[1:2] => 'el'
  - word[-1] => last character
  - **len**(word) => 5; length of the string
  - immutable: cannot assign to subscript

```
word = "Hello"
print(word[2])  # l
# Slicing [start:end:step]
print(word[1:3])  # el;
print(word[-1])  # o
print(word[2:-1])  # ll
print(word[::-1])  # Reverse the string
word[0] = 'h'  # error - cannot assign to subscript
```

- In python everything is object. There are some methods that can be applied to strings
  - E.g. join, find, replace, split, lower, upper and etc.
    - word = "Hello", word = word.replace('e', 'E') => HEllo
  - To get list of all the functions, print **dir('str')**

Introduction to Python - User Input

- To get inputs from the user use the input function.
  - If you need you can cast the input using int and float
- Command line arguments
  - You can get list of arguments using **sys.argv**
- Conditions: if, elif and else
- Loops: for and while loops
- break and continue like C

```
name = input("What is your name? ")
age = int(input("How old are you? "))
print("{} is {} years old.".format(name, age))
```

```
import sys

print(sys.argv)

number = input("Enter a number between 1 and 10: ")

if number.isnumeric():
    number = int(number)
    if (number < 1):
        print("Number is too small!")
    elif (number > 10):
        print("Number is too big!")
    else:
        print("You chose the number {}".format(number))
else:
    print("The entered data was not a number!")
```

```
for value in range(1, 20, 2):
    print(value**2)
```

```
number = 10
while number > 0:
    print("{:02}) Hello World!".format(number))
    number -= 1
```

Introduction to Python - User Input

- Equivalent of the ternary operator in C: ***d = a if c == True else b***
- pass is used to do nothing
- for/else and while/else: The else clause is executed after the loop completes normally

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            break
    else:
        print(n, 'is prime')
```

```
for n in range(2, 10):
    x = 2
    while x < n:
        if n % x == 0:
            break
        x += 1
    else:
        print(n, 'is prime')
```

```
while True:
    pass # Do nothing
```

Introduction to Python - Sequences

- **List** is one of the most useful data structure.
- A list is like a container to group multiple variable together.
- Lists are similar to arrays in C
  - Unlike arrays, elements in a list can have different types
  - E.g. my_list = [1, "python", 3.1415, True]

```
my_list = [1, "python", 3.1415, False, None]
my_list[0] = 2
my_list.remove("python")
del my_list[0]
my_list.append("language")
print(my_list[::-1])
```

- To get access to the elements we can use indices. E.g. my_list[0] = 0
- Functions like **len**(), **min**(), **max**() can be used with lists. E.g. len(my_list) => 4
- A list is mutable and we can modify it in different ways
  - Slicing is possible; my_list[start:end:step].
  - There are so some functions to modify a list. E.g. .append, .clear, .remove, .sort, .pop, etc.
  - E.g. nums = **list**(**range**(0, 100, 5)), nums[-1] = 0, nums.extend([1000, 2000]), print(nums)

Introduction to Python - Sequences

- It is possible to add two lists. E.g. list1 = [0, 1, 2] + [True, None, "Python"]
- When we assign a list to a variable a reference is copied, not the list.
  - E.g. list2 = list1; now list1 and list2 point to the same list
- Slicing returns copy of a subset. E.g. list2 = list1[1:4] or list2 = list1[:], list2 is a copy
- We can use for loops on lists (the in operator is used)

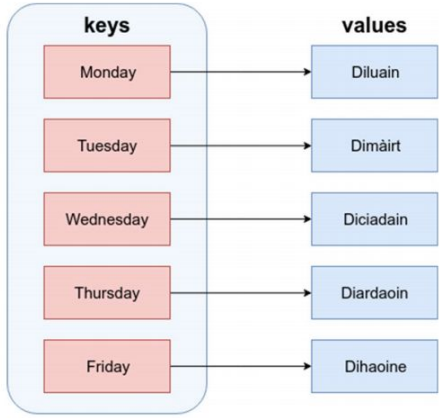
```
for v in range(0, 50, 5):
    print(v)
```

```
names = ["Maria", "Eric", "Lars", "Eva"]
for name in names:
    print("names[{}] = {}".format(names.index(name), name))
```

```
names = ["Maria", "Eric", "Lars", "Eva"]
for name in names:
    print(name)

for i, v in enumerate(names):
    print("names[{}] = {}".format(i, v))
```

Introduction to Python - Sequences

- A **Tuple** is an immutable list. Means that it can not be changed after creation.
- Tuples are created using () operator. E.g. t1 = (1, 2, 3, True, "Hello")
  - t1 is immutable. E.g. t1[2] = 10 # TypeError: 'tuple' object does not support item assignment
- Possible to add tuples. E.g. t1 = (1, 2, (3, "hello"), "World"), t1 = t1 + ('a', 'b')
- **Dictionaries** are similar to actual dictionaries
  - Combination of 2 lists - keys and values
  - Keys are used to get access the values
  - Each value is mapped to a unique key
  - Are created using {} operator. For examples:




```
d1 = {"Lars": "teacher", "Eva": "student", 42: "number"}
d2 = {"integers": [1, 2, 3, 4, 5], "primes": [2, 3, 5], "comments": {1: "first", 2: "second"}}
```

Introduction to Python - Sequences

- A dictionary is defined as comma separated of **key**:**value** pairs:
- Dictionary properties
  - Values are mapped to keys
  - Values are accessed by their key
  - Keys are unique and are immutable
  - A value cannot exist without a key

```
person_ages = {"Maria": 23, "Erik": 31, "Eva": 40}

for name in person_ages.keys():
    print("{} is {} years old".format(name, person_ages[name]))

for name in person_ages:
    print("{} is {} years old".format(name, person_ages[name]))

for name, age in person_ages.items():
    print("{} is {} years old".format(name, age))
```

- Values are accessed by their keys. E.g. mydict["Eva"] => "student"
- A dictionary can be modified in different ways
  - E.g. mydict["Eva"] = "programmer", mydict.clear(), mydict.get("Eva"), mydict.pop("Eva"), etc.
- It is possible to obtain only the keys or values of a dictionary: mydict.**keys**() and mydict.**values**()
- When we assign a dictionary to a variable, a reference is copied.

Introduction to Python - Sequences

- To copy a dictionary you need to **import copy** and use copy
- To get length of a dictionary you can use **len**().

```
import copy

mydict = person_ages.copy()
del person_ages
print(len(mydict), mydict)
```

- A **set** is a list that can't contain duplicate items
  - Similar functionality to lists
  - Can't be indexed or sliced
  - Can be created using {} or you can convert a list to a set
  - E.g. myset = {1, True, "Python", None, 2.5}

```
myset = {1, 2, 3}
print(len(myset), myset)
print(type(myset))
```

```
myset = {1, 2, 3, 3}
print(len(myset), myset)
print(type(myset))
```

```
myset = set([1, 2, 3, 3])
print(len(myset), myset)
print(type(myset))
```

```
s = myset.copy()
del myset

for v in s:
    print(v)

for i, v in enumerate(s):
    print(i, v)
```

Introduction to Python - Exit Codes

- Exit codes are use to tell the system
  - If the program was terminated successfully or if there was an error.
- Usually the exit code is ignored by the system unless we want to check and use it
- The default exit code is 0 which means no errors
- In C the exit code is set either by the return value of main() or by calling exit(…)
- In Python we use sys.exit(…) to immediately exit with a return code
- We need to **import sys**. E.g. exit(2)
- In bash we can check the return code of the last terminated program

  $ **./example.py**

  $ **echo $?** # It prints the exit code to the terminal

Introduction to Python - Comprehensions

- Comprehension is a short and elegant way to
  - Create a new sequence using an already existing sequence.
- Python supports 4 types of comprehensions: List, Dictionary, Set and Generator
- List comprehensions basic syntax
  - *output_list = [output_exp for var in input_list [if var satisfies this condition]]*
- Dictionary comprehension is similar to list comprehension
  - *new_dict = {key:value for (key, value) in iterable [if (key, value satisfy this condition)]}*

```
input_list = [1, 2, 3, 4, 5, 6, 7]
dict1 = {var: var ** 3 for var in input_list if var % 2 != 0}
squares = {i: i * i for i in range(10)}
list1 = [x for x in range(5)]
list2 = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
dict2 = {key: value for (key, value) in zip(list1, list2)}
```

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
list1 = [x**2 for x in input_list]
list2 = [x for x in input_list if x % 2 == 0]
list3 = [x**3 for x in range(20) if x % 2 == 0 and x % 3 == 1]
list4 = [y for y in range(100) if y % 2 == 0 if y % 5 == 0]
sentence = 'the rocket came back from mars'
list5 = [i for i in sentence if i in 'aeiou']
```

Introduction to Python - Comprehensions

- Set comprehension is similar to list comprehension
  - The only difference between them is that set comprehensions use curly brackets { }
- **Generator** Comprehension
  - Is very similar to list comprehensions
  - Instead of [], **()** is used to create a generator
  - Is very memory efficient
    - Unlink list comprehensions, generators don't allocate memory for the whole list
    - Very useful when we deal with big lists

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
set1 = {x**2 for x in input_list}
set2 = {x for x in input_list if x % 2 == 0}
set3 = {x**3 for x in range(20) if x % 2 == 0 and x % 3 == 1}
set4 = {y for y in range(100) if y % 2 == 0 if y % 5 == 0}
print(set1, set2, set3, set4)
```

```
input_list = [x for x in range(50)]
gen = (var + 3 for var in input_list)
print("Output:", end=' ')
for var in gen:
    print(var, end=' ')
```

```
total = sum(i * i for i in range(100000))
print("Sum =", total)
```

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
gen = (var for var in input_list if var % 2 == 0)

print("Output:", end=' ')
for var in gen:
    print(var, end=' ')
print()
```

Introduction to Python - Exceptions and Error Handling

- Exceptions are used for error handling during runtime.
  - Errors detected during execution are called exceptions
    - E.g. When a number is divided by zero
  - When such errors occur, the interpreter stops the current process and raises an exception to handle. If not handled, the program will crash.
  - There are so many types of exception and every exception in Python is an object
    - E.g. ZeroDivisionError, FileNotFound and etc.
    - To read more about exceptions look at Errors and Exceptions
- We can **catch** exceptions and even we can **raise** exceptions
- Possible to catch multiple exceptions

```
except (RuntimeError, TypeError, NameError, ValueError) as err:
    print(err)
```

```
a, b = 1, 0
try:
    c = a / b
except ZeroDivisionError as err:
    print(err)
```

```
try:
    number = int(input("Enter an even number: "))
    if number % 2 != 0:
        raise Exception("The number is not even!")
except ValueError as err:
    print("The input is not a number: ", err)
except Exception as err:
    print(err)
```

Introduction to Python - Exceptions and Error Handling

- The try statement works as follows:
  - First, the *try clause* (the statement(s) between the try and except keywords) is executed.
  - If no exception occurs, the except clause is skipped and execution of the try statement is finished.
  - If an exception occurs during execution of the try clause, the rest of the clause is skipped
    - If its type matches the exception named after the except keyword, the except clause is executed
      - Then execution continues after the try statement.
    - If an exception occurs which does not match the exception named in the except clause
      - It is passed on to outer try statements
      - If no handler is found, it is an unhandled exception and execution stops
- It is possible to pass multiple args to raised exceptions
- It is possible to omit the exception name(s)

```
except:
    print("Unexpected error:")
```

```
try:
    raise Exception('aaa', 'bbb')
except Exception as excp:
    print(type(excp))    # the exception instance
    print(excp.args)     # arguments stored in .args
    print(excp)
```

Introduction to Python - Exceptions and Error Handling

- The try … except statement has an optional else clause
  - If it exists, it must follow all except clauses.
  - It is useful for code that must be executed if the try clause does not raise an exception.

```
try:
    f = open('file.txt', 'r')
except OSError:
    print('cannot open file')
else:
    f.close()
```

- The try … except statement has an optional finally clause also
  - It is used to define clean-up actions that must be executed under all circumstances
  - If it exists, it will be executed as the last task before the try statement completes
  - The finally clause runs whether or not the try statement produces an exception.

```
try:
    raise KeyboardInterrupt
except:
    print('KeyboardInterrupt')
finally:
    print('Goodbye, world!')
```

```
try:
    result = x / y
except ZeroDivisionError:
    print("division by zero!")
else:
    print("result is", result)
finally:
    print("executing finally clause")
```

Introduction to Python - File I/O

- File handling is similar to C file handling
  - We open a file in a mode and get a handle to the file
  - We can use the handle to read and write data.
  - When we are done we must close the file.
- Binary and text mode may be added to these modes, e.g. 'rb', 'wt'
- When opening a file in binary mode, all reads return bytes and you can only write bytes
- Similarly for text mode reads and writes uses strings

Mode	Description
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	create a new file and open it for writing
'a'	open for writing, appending to the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a file for updating (reading and writing)

Introduction to Python - File I/O

- Functions
  - open - is used to open a file
  - .read() – read data until end of file
  - .read(10) – read at most 10 bytes of data
  - .readline() – read the next line
  - .readlines() – read all lines and return them in a list
  - .write(data) – writes data to file
  - .writelines(lines) – writes list of lines to file
  - .tell() – returns current position in file
  - .seek(*offset[, whence]*) – change position in file
    - It is like fseek in C, you pass an **offset** from **whence** to the function
    - **whence** is optional; it can be **SEEK_SET**, **SEEK_CUR** or **SEEK_END**

```
try:
    file = open("file.txt", "wt")
    file.write("Hello World!\n" * 10)
except Exception as err:
    print(err)
```

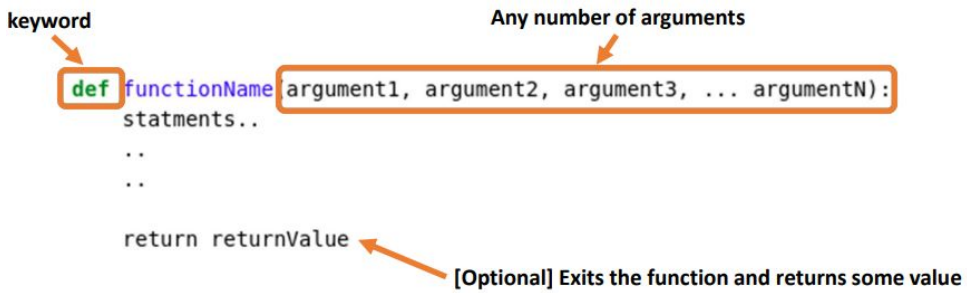
```
try:
    file = open("file.txt", "r+t")
    lines = file.readlines()
    for index, line in enumerate(lines):
        lines[index] = line.replace("World", "Stefan")
    file.seek(0)
    file.truncate(0)  # truncate does not move the file position
    file.writelines(lines)
    file.close()
except OSError as err:
    print(err)
```

Using **with** to ensure the file is closed

```
with open("file.txt", "wt") as file:
    try:
        file.write("Hello World!\n" * 10)
    except:
        print("Failed to write to the file!")
```

Introduction to Python - Functions

- A function is a block of code which only runs when it is called.
- Data can be passed to a function and a function can return data as a result.
- To define a function the def keyword is used.
- Parameters are specified after the function name, inside the parentheses
- Possible to specify the types
- By default, a function must be called with the correct number of arguments.
  - E.g. hello_func("World") is ok.
  - E.g. hello_func() or hello_func("A", 21) => error




```
def hello_func(name: str) -> None:
    print("Hello {}!".format(name))

def hello_func(name):
    print("Hello {}!".format(name))

hello_func("World")
```

Introduction to Python - Functions

- Unknown Number of Arguments, ***args**
  - If the number of arguments is unknown, add a * before the parameter name
  - The function will receive a tuple of arguments

```
def my_function(*args):
  print("The youngest child is " + args[2])

my_function("Emil", "Tobias", "Linus")
```

- Keyword Arguments
  - Possible to send arguments with the **key = value** syntax.
  - In this case the order of the arguments does not matter.

```
def func(child3, child2, child1):
  print("The youngest child is " + child3)

func(child1 = "Emil", child2 = "Tobias", child3 = Linus")
```

- Unknown Number of Keyword Arguments, ****kwargs**
  - If the number of keyword arguments is unknown, add ** before the parameter name
  - The function will receive a dictionary of arguments

```
def func(**kwargs):
  print("His last name is " + kwargs["lname"])

func(fname="Emil", lname="Larsson")
```

Introduction to Python - Functions

- Default Parameter Value
  - If we call the function without argument, it uses the default value

```
def func(country = "Norway"):
  print("I am from " + country)

func("Sweden")
func()
```

- To let a function returns a value, use a return statement
- Python supports function recursion, and a defined function can call itself

```
def func(x):
  return 5 * x

print(func(3))
print(func(5))
```

- Possible to have inner functions
  - Functions in functions
  - Inner functions are not global

```
def factorial(n):
  return (1 if n == 1 else n*factorial(n-1))

print(factorial(5))
```

```
def func(name: str):
  print(name)
  # Inner function
  def ufunc():
    print(name.upper())

  ufunc()

func("Stefan")
```

- A **lambda** function is a small anonymous function
- A **lambda** function can take any number of arguments, but can only have one expression.
- Syntax: *lambda arguments : expression*

Introduction to Python - Functions

- In a lambda function the expression is executed and the result is returned
- The power of lambda is better shown when you use them as an anonymous function inside another function.
- A variable created inside a function belongs to the **local scope** of that function, and can only be used inside that function.
- A local variable is available for any function inside the function
- A variable created in the main body of the Python code is a **global** variable and belongs to the global scope and it is available in any scope
- The global keyword makes a variable **global**.
- Use the global keyword if you want to make a change to a global variable inside a function

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

x = lambda a, b: a * b
print(x(5, 6))
```

```
def multiplier(n):
    return lambda a: a * n

doubler = multiplier(2)
print(doubler(11))
tripler = multiplier(3)
print(tripler(11))
```

```
y = 200
def func():
    x = 300  # Local variable
    global y
    y = 300
    def inner_func():
        print(x + y)
    inner_func()

func()
print(y)
```

Introduction to Python - Modules

- **Modular programming** is the process of breaking a large task into separate, smaller, more manageable subtasks or modules.
- Individual modules as building blocks can be linked together to create a larger application.
- Modularity provides simplicity, maintainability, reusability and scoping
- Functions, modules and packages in Python are used for code modularization
- In Python there are so many built-in modules and also you can create your own
- You can import modules to your code using import. E.g import sys
- A module is a file containing Python definitions and statements
  - A collection of variables, functions and classes
  - The file name is the module name + .py. E.g. fibo.py, fibo is the module name
  - The definitions of objects can be imported

Introduction to Python - Modules

- Within a module, the module's name is available in the global variable **__name__**
  - A module can also be executed as a standalone script.
    - In this case the value of __name__ is "**__main__**"
- In different ways we can import definitions in a module

```
# An example script

import fibo

fibo.fib(10)
print(fibo.fib2(10))
```

```
# An example script

from fibo import fib, fib2

fib(10)
print(fib2(10))
```

```
try:
    # Non-existent module
    import baz
except ImportError:
    print('Module not found')
```

```
# An example script

import fibo as alt_name

alt_name.fib(10)
print(alt_name.fib2(10))
```

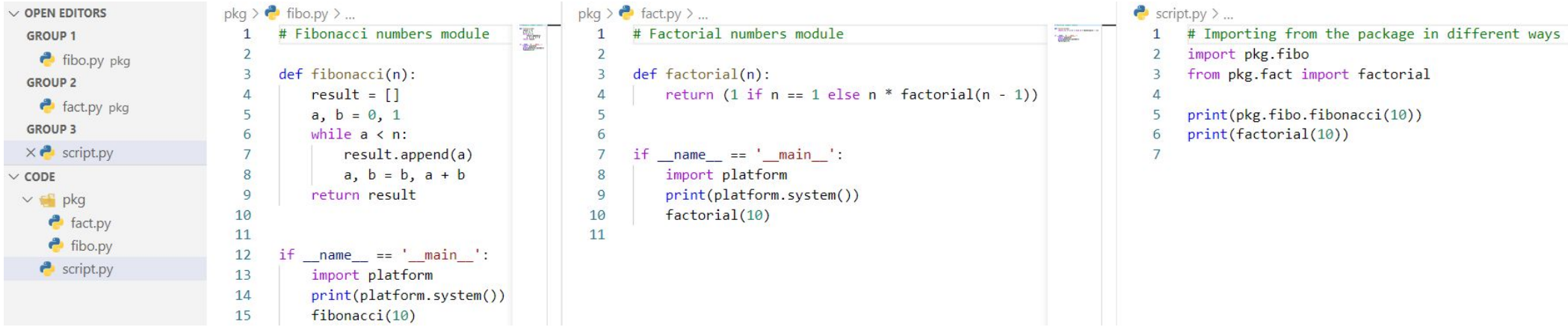
```
# An example script

from fibo import fib as f1, fib2 as f2

f1(10)
print(f2(10))
```

```
# An example script

from fibo import *

fib(10)
print(fib2(10))
```

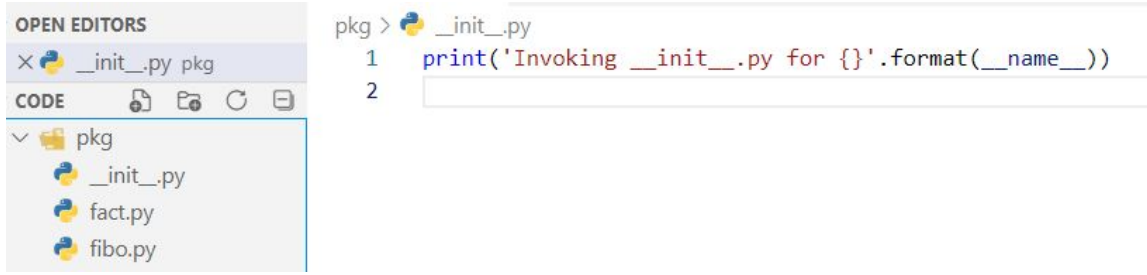
```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a + b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a + b
    return result

if __name__ == '__main__':
    import platform
    print(platform.system())
    fib(10)
```

Introduction to Python - Packages

- Packages are structured as directories of modules using **dot notation**



- If a file named **__init__.py** is present in a package directory
  - It is invoked when the package or a module in the package is imported
  - Usually it is used for package initialization code



Introduction to Python

- Some useful links
  - Python Tutorial for Beginners
  - Python for Beginners
  - The Python Tutorial
  - Learn Python - Full Course for Beginners
  - Learn Python - Free Interactive Python tutorial
  - Python Cheat sheet!