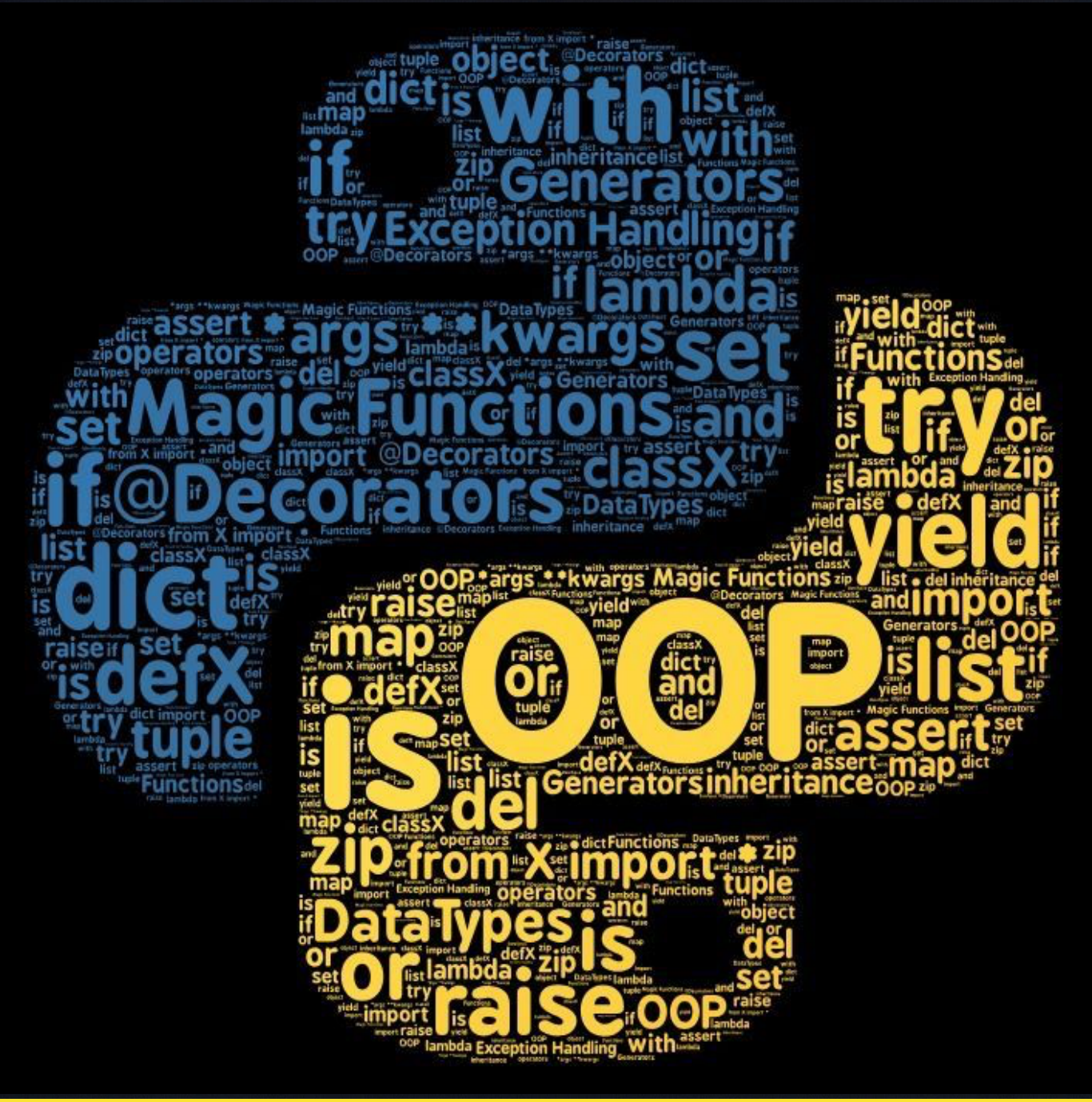


PYTHON PROGRAMMING

Assoc. Prof. Dr. Bora Canbula

Manisa Celal Bayar University
Department of Computer Engineering

https://github.com/canbula/PythonProgramming/ gkdtw9f

Python Programming

Instructor

Assoc. Prof. Dr.
Bora CANBULA

Phone

0 (236) 201 21 08

Email

bora.canbula@cbu.edu.tr

Office Location

Dept. of CENG

Office C233

Office Hours

4 pm – 5 pm, Mondays

Course Overview

Python Programming (Teams Code: gkdtw9f)

This course is about basic concepts in Python Programming. Students will learn the data structures, functions, classes, and the special structures of Python. They will also learn how to create basic desktop applications with a GUI, web applications as APIs, basic games as projects. In this course, we will use GitHub actions for assignments.

Required Text

Advanced Guide to Python 3 Programming, Springer, *John Hunt*

Python for Everybody, PY4E, *Charles Severance*

Course Materials

- Python 3.x (Anaconda is preferred)
- Jupyter Notebook from Anaconda
- Pycharm from JetBrains / Microsoft Visual Studio Code
- PC with a Linux distro or a Linux terminal in Windows 10/11.

Course Schedule

Week	Subject	Week	Subject
01	Basic Concepts in Python	08	Desktop Programming
02	Sequences	09	Desktop Programming
03	Functions	10	Web Programming
04	Decision Structures	11	Web Programming
05	Loops	12	Game Programming
06	Classes	13	Game Programming
07	Special Structures	14	Student Projects

Python Programming

Instructor

Assoc. Prof. Dr.
Bora CANBULA

Phone

0 (236) 201 21 08

Email

bora.canbula@cbu.edu.tr

Office Location

Dept. of CENG

Office C233

Office Hours

4 pm – 5 pm, Mondays

Course Overview

Python Programming (Teams Code: gkdtw9f)

This course is about basic concepts in Python Programming. Students will learn the data structures, functions, classes, and the special structures of Python. They will also learn how to create basic desktop applications with a GUI, web applications as APIs, basic games as projects. In this course, we will use GitHub actions for assignments.

Required Text

Advanced Guide to Python 3 Programming, Springer, *John Hunt*

Python for Everybody, PY4E, *Charles Severance*

Course Materials

- Python 3.x (Anaconda is preferred)
- Jupyter Notebook from Anaconda
- Pycharm from JetBrains / Microsoft Visual Studio Code
- PC with a Linux distro or a Linux terminal in Windows 10/11.

Project Themes

Sports

Economy

Health

Project Examples

Sports

- Predicting Game Outcomes
- Injury Risk Prediction
- Player Market Value Prediction
- Fan Engagement Analysis
- Athlete Performance Comparison

Economy

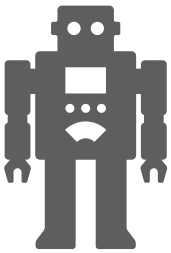
- Stock Market Prediction
- Cryptocurrency Price Prediction
- Consumer Spending Analysis
- Predicting Unemployment Rates
- Credit Scoring Model

Health

- Disease Outbreak Prediction
- Personalized Health Recommendations
- Health Risk Prediction
- Hospital Readmission Prediction
- Nutritional Deficiency Prediction

Natural Languages vs. Programming Languages

A language is a tool for expressing and recording thoughts.



Computers have their own language called **machine** language. Machine languages are created by humans, no computer is currently capable of creating a new language. A complete set of known commands is called an instruction list (IL).

The difference is that human languages developed naturally. They are still evolving, new words are created every day as old words disappear. These languages are called **natural** languages.

Elements of a Language

- **Alphabet** is a set of symbols to build words of a certain language.
- **Lexis** is a set of words the language offers its users.
- **Syntax** is a set of rules used to determine if a certain string of words forms a valid sentence.
- **Semantics** is a set of rules determining if a certain phrase makes sense.

Machine Language vs. High-Level Language

The IL is the alphabet of a machine language. It's the computer's mother tongue.

High-level programming language enables humans to write their programs and computers to execute the programs. It is much more complex than those offered by ILs.

A program written in a high-level programming language is called a **source code**. Similarly, the file containing the source code is called the **source file**.

Compilation vs. Interpretation

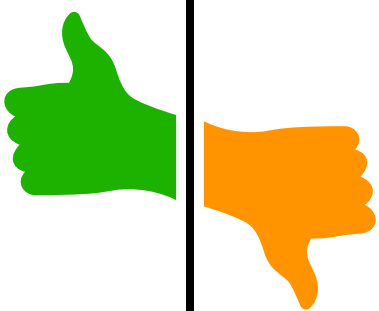
There are two different ways of transforming a program from a high-level programming language into machine language:

Compilation: The source code is translated once by getting a file containing the machine code.

Interpretation: The source code is interpreted every time it is intended to be executed.

Compilation

- The execution of the translated code is usually faster.
- Only the user has to have the compiler. The end user may use the code without it.
- The translated code is stored using machine language. Your code are likely to remain your secret.



- The compilation itself may be a very time-consuming process
- You have to have as many compilers as hardware platforms you want your code to be run on.

Interpretation

- You can run the code as soon as you complete it, there are no additional phases of translation.
- The code is stored using programming language, not machine language. You don't compile your code for each different architecture.

- Don't expect interpretation to ramp up your code to high speed
- Both you and the end user have the interpreter to run your code.

What is Python?

Python is a widely-used, interpreted, object-oriented, and high-level programming language with dynamic semantics, used for general-purpose programming.

Python was created by Guido van Rossum. The name of the Python programming language comes from an old BBC television comedy sketch series called Monty Python's Flying Circus.

Python Goals

- an **easy and intuitive** language just as powerful as those of the major competitors
- **open source**, so anyone can contribute to its development
- code that is as **understandable** as plain English
- **suitable for everyday tasks**, allowing for short development times

Why Python?

- easy to learn
- easy to teach
- easy to use
- easy to understand
- easy to obtain, install and deploy

Why not Python?

- low-level programming
- applications for mobile devices

Python Implementations

An implementation refers to a program or environment, which provides support for the execution of programs written in the Python language.

- **CPython** is the traditional implementation of Python and it's most often called just "Python".
- **Cython** is a solution which translate Python code into "C" to make it run much faster than pure Python.
- **Jython** is an implementation follows only Python 2, not Python 3, written in Java.
- **PyPy** represents a Python environment written in Python-like language named RPython (Restricted Python), which is actually a subset of Python.
- **MicroPython** is an implementation of Python 3 that is optimized to run on microcontrollers.

Start Coding with Python

- **Editor** will support you in writing the code. The Python 3 standard installation contains a very simple application named IDLE (Integrated Development and Learning Environment).
- **Console** is a terminal in which you can launch your code.
- **Debugger** is a tool, which launches your code step-by-step to allow you to inspect it.

first.py

```
print("Python is the best!")
```

https://forms.office.com/r/z2XZbbypBR

Function Name
A function can cause some effect
or evaluate a value, or both.

Where do functions come from?
- From Python itself
- From modules
- From your code

first.py

```
print("Python is the best!")
```

Argument
- Positional arguments
- Keyword arguments

`print``(*objects, sep=' ', end='\n', file=None, flush=False)`

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file*, and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Output buffering is usually determined by *file*. However, if *flush* is true, the stream is forcibly flushed.

Literals

A literal is data whose values are determined by the literal itself. Literals are used to encode data and put them into code.

literals.py

```
print("7")
print(7)
print(7.0)
print(7j)
print(True)
print(0b10)
print(0o10)
print(0x10)
print(7.4e3)
```

- String
- Integer
- Float
- Complex
- Boolean
- Binary
- Octal
- Hexadecimal
- Scientific Notation

Basic Operators

An operator is a symbol of the programming language, which is able to operate on the values.

Multiplication

```
print(2 * 3)      Integer
print(2 * 3.0)    Float
print(2.0 * 3)    Float
print(2.0 * 3.0)  Float
```

Division

```
print(6 / 3)      Float
print(6 / 3.0)    Float
print(6.0 / 3)    Float
print(6.0 / 3.0)  Float
```

Exponentiation

```
print(2**3)       Integer
print(2**3.0)     Float
print(2.0**3)     Float
print(2.0**3.0)   Float
```

Floor Division

```
print(6 // 3)     Integer
print(6 // 3.0)   Float
print(6.0 // 3)   Float
print(6.0 // 3.0) Float
```

Modulo

```
print(6 % 3)      Integer
print(6 % 3.0)    Float
print(6.0 % 3)    Float
print(6.0 % 3.0)  Float
```

Addition

```
print(-8 + 4)     Integer
print(-4.0 + 8)   Float
```

Operator Priorities

An operator is a symbol of the programming language, which is able to operate on the values.

priorities.py

```
print(9 % 6 % 2)
print(2**2**3)
print(2 * 3 % 5)
print(-3 * 2)
print(-2 * 3)
print(-(2 * 3))
```

- + (unary)
- - (unary)
- ** (right-sided binding)
- *
- /
- //
- % (left-sided binding)
- + (binary)
- - (binary)

Variables

Variables are symbols for memory addresses.

Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

Built-in Functions

A	E	L	R
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
anext()			round()
any()	F	M	
ascii()	filter()	map()	S

hex(*x*)

Convert an integer number to a lowercase hexadecimal string prefixed with "0x". If *x* is not a Python `int` object, it has to define an `__index__()` method that returns an integer. Some examples:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

classmethod()	help()	ord()	type()
compile()	hex()		
complex()		P	V
	I	pow()	vars()
D	id()	print()	

id(*object*)

Return the "identity" of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.

https://docs.python.org/3/library/functions.html

Identifier Names

For variables, functions, classes etc. we use identifier names. We must obey some rules and we should follow some naming conventions.

- Names are case sensitive.
- Names can be a combination of letters, digits, and underscore.
- Names can only start with a letter or underscore, can not start with a digit.
- Keywords can not be used as a name.

keyword — Testing for Python keywords

Source code: Lib/keyword.py

This module allows a Python program to determine if a string is a keyword or soft keyword.

`keyword.`**`iskeyword`**`(s)`
Return `True` if *s* is a Python keyword.

`keyword.`**`kwlist`**
Sequence containing all the keywords defined for the interpreter. If any keywords are defined to only be active when particular `__future__` statements are in effect, these will be included as well.

`keyword.`**`issoftkeyword`**`(s)`
Return `True` if *s* is a Python soft keyword.

New in version 3.9.

`keyword.`**`softkwlist`**
Sequence containing all the soft keywords defined for the interpreter. If any soft keywords are defined to only be active when particular `__future__` statements are in effect, these will be included as well.

New in version 3.9.

https://forms.office.com/r/y6xQaXSL1F

Your First Homework

Week01/info_firstname_lastname.py

- ☑ A string variable with the name student_id that contains your student id.
- ☑ A string variable with the name full_name that contains your full name.

Week02/types_firstname_lastname.py

- ☑ An integer with the name: my_int
- ☑ A float with with the name: my_float
- ☑ A boolean with the name: my_bool
- ☑ A complex with the name: my_complex

Rules for your pull requests

- ✓ Please run your code first in your computer, do not submit codes with syntax errors.
- ✓ Submit your code to WeekXX folder. WeekXX/hw is for me to move accepted works.
- ✓ If a change is requested, please edit the existing pull request, don't open a new one.

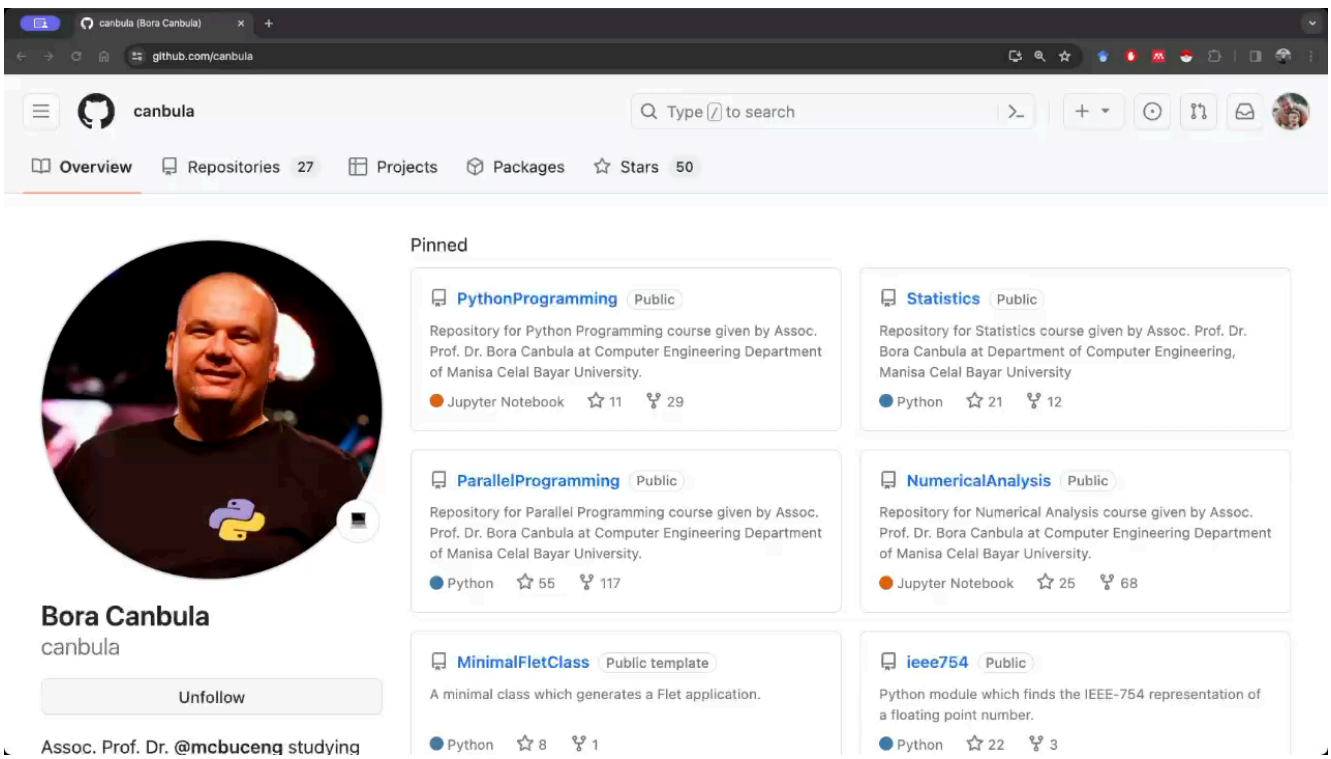
Your First Homework

Week01/info_firstname_lastname.py

- ☑ A string variable with the name student_id that contains your student id.
- ☑ A string variable with the name full_name that contains your full name.

Week02/types_firstname_lastname.py

- ☑ An integer with the name: my_int
- ☑ A float with with the name: my_float
- ☑ A boolean with the name: my_bool
- ☑ A complex with the name: my_complex



Equality & Identity & Comparison

> The operators `is` and `is not` test for an object's identity: `x is y` is true if and only if *x* and *y* are the same object. An Object's identity is determined using the `id()` function. `x is not y` yields the inverse truth value. [4]

Equality

```
a = 4
print(a == 4) ✓
print(id(a) == id(4)) ✓
print(a is 4) ✓
print(id(a) is id(4)) ✗
```

Left- or Right-sided?

```
x, y, z = 0, 1, 2
print(x == y == z) ✗
print(x == (y == z)) ✓
print((x == y) == z) ✗
```

Inequality

```
print(1 != 2) ✓
print(not 1 == 2) ✓
```

Updated Priority Table

Operator	Type
+, -	unary
**	binary
*, /, //, %	binary
+, -	binary
<, <=, >, >=	binary
!=, ==	binary

Comparison

```
print(1 < 2) ✓
print(1 <= 2) ✓
print(1 > 2) ✗
print(1 >= 2) ✗
```

Chaining

```
print(1 < 2 < 3) ✓
print(1 < 2 > 3) ✗
print(1 < 2 >= 3) ✗
print(1 < 2 <= 3) ✓
```

QUESTION

Using one of the comparison operators in Python, write a simple two-line program that takes the parameter **n** as input, which is an integer, prints **False** if **n** is less than **100**, and **True** if **n** is greater than or equal to **100**.

```
n = int(input())
print(n >= 100)
```

Conditional Execution

if statement

```
if n >= 100:
    print("The number is greater than or equal to 100.")
elif n < 0:
    print("The number is negative.")
else:
    print("The number is less than 100.")
```

Ternary Operator

```
msg = "The number is greater than or equal to 100." if n >= 100 else "The number is less than 100."
print(msg)
```

Loops

QUESTION

- The program generates a random number between 1 and 10.
- The user is asked to guess the number.
- The user is given feedback if the guess is too low or too high.
- The user is asked to guess again until the correct number is guessed.

```
r = random.randint(1, 10)
answer = False
while not answer:
    n = int(input("Enter a number: "))
    if n == r:
        print("You guessed it right!")
        answer = True
    elif n < r:
        print("Try a higher number.")
    else:
        print("Try a lower number.")
```

QUESTION

- The user is asked to enter a number.
- The program prints the numbers from 0 to n-1.

```
n = int(input("Enter a number: "))
for i in range(n):
    print(i, end=" ")
```

break and continue

```
n = int(input("Enter a number: "))
for i in range(10):
    if i < n:
        print("The number is not found:", i)
        continue
    if i == n:
        print("The number is found:", i)
        break
```

```
r = random.randint(1, 10)
while True:
    n = int(input("Enter a number: "))
    if n == r:
        print("You guessed it right!")
        break
    elif n < r:
        print("Try a higher number.")
    else:
        print("Try a lower number.")
```

class **range**(*stop*)
class **range**(*start*, *stop*[, *step*])

The arguments to the range constructor must be integers (either built-in `int` or any object that implements the `__index__()` special method). If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. If *step* is zero, `ValueError` is raised.

For a positive *step*, the contents of a range `r` are determined by the formula `r[i] = start + step*i` where `i >= 0` and `r[i] < stop`.

For a negative *step*, the contents of the range are still determined by the formula `r[i] = start + step*i`, but the constraints are `i >= 0` and `r[i] > stop`.

start
The value of the *start* parameter (or `0` if the parameter was not supplied)

stop
The value of the *stop* parameter

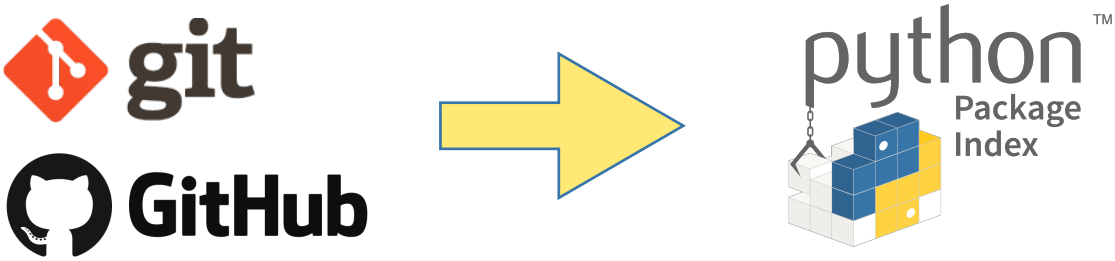
step
The value of the *step* parameter (or `1` if the parameter was not supplied)

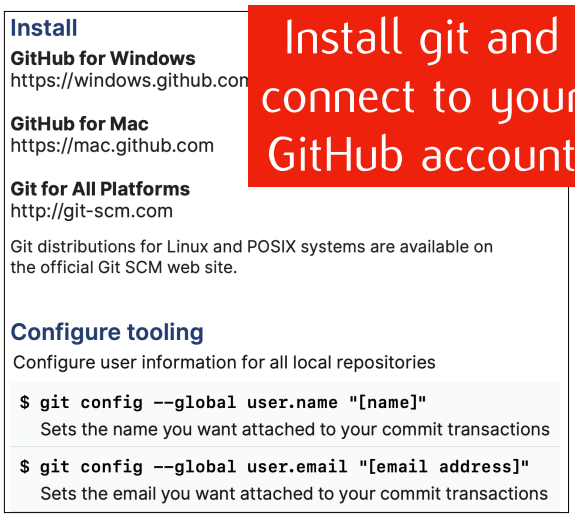
The advantage of the `range` type over a regular `list` or `tuple` is that a `range` object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the `start`, `stop` and `step` values, calculating individual items and subranges as needed).

Can we use while/for with else?

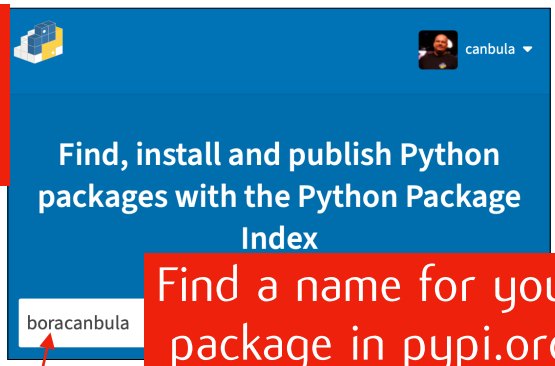
Packaging Python Modules

Steps that are required for packaging our modules and make them available to be installed via pip.

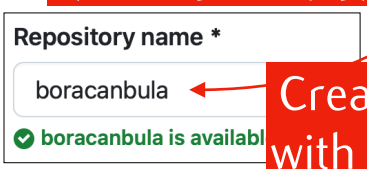




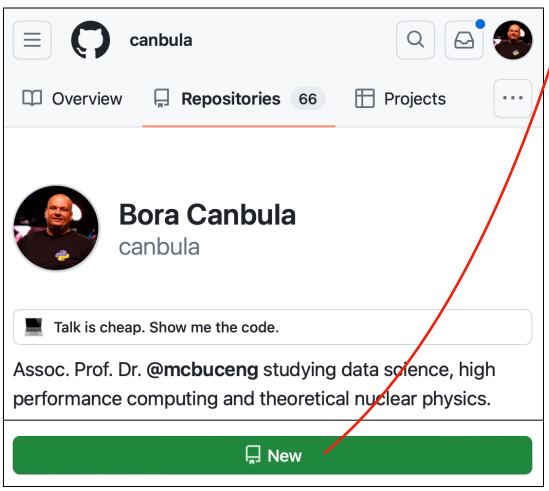
Install git and connect to your GitHub account

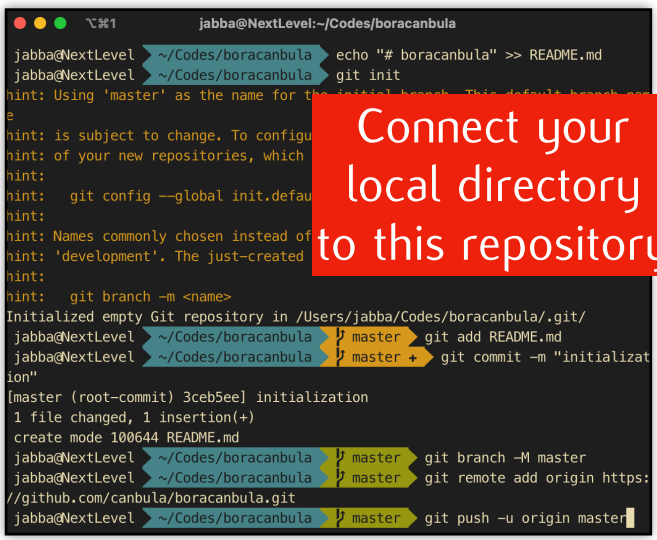


Find a name for your package in pypi.org



Create a repo with this name

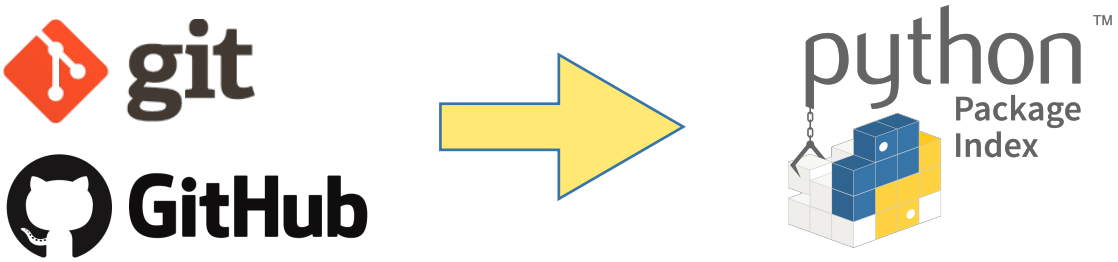




Connect your local directory to this repository

Packaging Python Modules

Steps that are required for packaging our modules and make them available to be installed via pip.



Folder Structure

```
boracanbula
├── dist
├── src
│   └── boracanbula
│       ├── __init__.py
│       └── statistics.py
├── tests
│   └── test_statistics.py
├── LICENSE
├── pyproject.toml
└── README.md
```

An empty directory

Directory for source codes

Directory using the name with your package

An empty Python file

The Python file that includes a function or a class (use an appropriate file name)

Directory for tests

A Python file which its name starts with test_

A text file that includes a license

A markdown file for the detailed description of your project

A text file with the .toml extension that includes the configuration details of your project

Packaging Python Modules

statistics.py

```python
def mean(x: list) -> float:
    """Calculate the mean of a list of numbers."""
    return sum(x) / len(x)
```

test_statistics.py

```python
import sys
sys.path.append(".")

import pytest
from src.boracanbula import statistics

def test_mean():
    assert statistics.mean([1, 2, 3, 4, 5]) == 3
```

LICENSE

```
MIT License

Copyright (c) 2024 Bora Canbula

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

README.md

```markdown
# boracanbula
This is a test package
for the PyPi tutorial.
```

pyproject.toml

```toml
[build-system]
requires = ["setuptools", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "boracanbula"
version = "0.0.1"
authors = [
    { name = "Bora Canbula", email = "bora.canbula@cbu.edu.tr" },
]
description = "A simple package for calculating statistics."
readme = "README.md"
keywords = ["canbula", "statistics"]
requires-python = ">=3.6.0"
classifiers = [
    "Development Status :: 3 - Alpha",
    "Intended Audience :: Developers",
    "License :: OSI Approved :: MIT License",
    "Programming Language :: Python :: 3",
    "Operating System :: OS Independent",
]

[project.urls]
Homepage = "https://github.com/canbula/boracanbula"
```

Packaging Python Modules

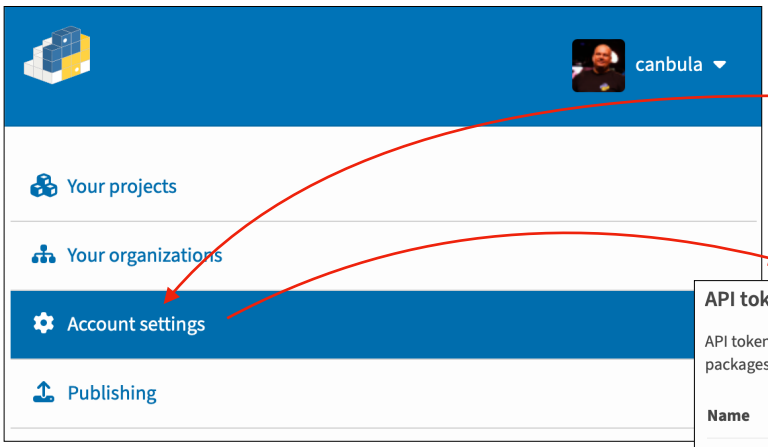
Be sure that you have the required modules

```
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade setuptools wheel
python3 -m pip install --upgrade build
python3 -m pip install --upgrade twine
```

Build

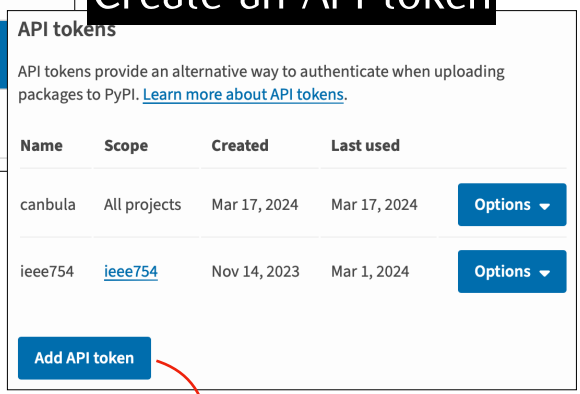
```
python3 -m build
```

dist
- boracanbula-0.0.1-py3-none-any.whl
- boracanbula-0.0.1.tar.gz



Go to your "Account settings" in pypi.org

Create an API token



.pypirc

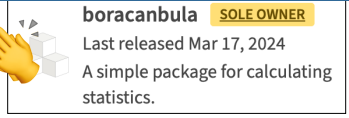
```
[distutils]
  index-servers =
    pypi
    boracanbula

[pypi]
  username = __token__
  password = pypi-AgEIcHlwaS5vcmcCJGQ2MWZiZDA0l
[boracanbula]
  repository = https://upload.pypi.org/legacy/
  username = __token__
  password = pypi-AgEIcHlwaS5vcmcCJGQ2MWZiZDA0l
```

Create a .pypirc file in your home directory with this token

Upload your package to PyPI

```
python3 -m twine upload --repository boracanbula dist/*
```

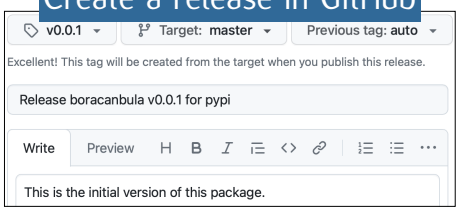
Check your package in PyPI



Install your package via pip

```
pip install --upgrade boracanbula
python3 -m pip install --upgrade boracanbula
py -m pip install --upgrade boracanbula
```

Create a release in GitHub



Functions

Functions are defined by using `def` keyword, name and the parenthesized list of formal parameters.

Function names should be lowercase, with words separated by underscores as necessary to improve the readability. PEP 8

Basic Function Definition

```
def function_name():
    pass
```

Input/Output Arguments

```
def fn(arg1, arg2):
    return arg1 + arg2
```

Default Values for Arguments

```
def fn(arg1=0, arg2=0):
    return arg1 + arg2
```

Type Hints and Default Values for Arguments

```
def fn(arg1: int = 0, arg2: int = 0) -> int:
    return arg1 + arg2
```

PEP 3107

Multiple Type Hints for Arguments

```
def fn(arg1: int|float, arg2: int|float) -> tuple[float, float]:
    return arg1 + arg2, arg1 * arg2
```

> Python 3.10

Lambda Functions

```
fn = lambda arg1, arg2: arg1 + arg2
```

Function Docstrings

```
def fn(arg1=0, arg2=0):
    """This function sums two numbers."""
    return arg1 + arg2
```

PEP 257

Docstrings

PEP 257

A docstring is a string literal that occurs as the first Statement in a module, function, class, or method definition. Such a docstring becomes the __doc__ special attribute of that object.

One-line Docstrings

```
def fn(arg1=0, arg2=0):
    """This function sums two numbers."""
    return arg1 + arg2
```

Multi-line Docstrings

```
def fn(arg1: int = 0, arg2: int = 0) -> int:
    """This function sums two numbers.

    Keyword arguments:
    arg1 -- first number (default 0)
    arg2 -- second number (default 0)
    Return: sum of arg1 and arg2
    """
    return arg1 + arg2
```

Docutils and Sphinx are tools to automatically create documentations

reST (reStructuredText) Format

```
def fn(arg1: int = 0, arg2: int = 0) -> int:
    """
    This function sums two numbers.

    :param arg1: The first number, def
    :type arg1: int
    :param arg2: The second number, de
    :type arg2: int
    :raises TypeError: Both arguments
    :return: The sum of the two number
    :rtype: int
    """
    if type(arg1) != int or type(arg2)
        raise TypeError("Both argument
    return arg1 + arg2
```

Google Format

```
"""
This function sums two numbers.

Args:
    arg1 (int): The first number, default is 0
    arg2 (int): The second number, default is 0

Returns:
    int: The sum of the two numbers

Raises:
    TypeError: Both arguments must be integers.
"""
```

Some other formats are Epytext (javadoc), Numpydoc, etc

Parameter Kinds

PEP 362

Kind describes how argument values are bound to the parameter. The kind can be fixed in the signature of the function.

Standard Binding: Positional-or-Keyword

```
def fn(arg1=0, arg2=0):
    return arg1 + arg2
```

```
fn(), fn(3), fn(3, 5), fn(arg1=3)
fn(arg2=5), fn(arg1=3, arg2=5)
```

Positional-or-Keyword & Keyword-Only

```
def fn(arg1=0, arg2=0, *, arg3=1):
    return (arg1 + arg2) * arg3
```

```
fn(), fn(3), fn(3, 5), fn(arg1=3), fn(arg2=5)
fn(arg1=3, arg2=5), fn(3, 5, arg3=2)
fn(arg1=3, arg2=5, arg3=2)
fn(arg3=2, arg1=3, arg2=5)
```

```
fn(3, 5, 2)
fn(arg3=2, 3, 5)
```

Positional-Only & Positional-or-Keyword & Keyword-Only

```
def fn(arg1=0, arg2=0, /, arg3=1, arg4=1, *, arg5=1, arg6=1):
    return (arg1 + arg2) * arg3 / arg4 * arg5**arg6
```

PEP 457

```
fn(), fn(3), fn(3, 5), fn(3, 5, 2), fn(3, 5, 2, 4)
fn(3, 5, arg3=2, arg4=4)
fn(3, 5, arg3=2, arg4=4, arg5=7, arg6=8)
fn(3, 5, 2, 4, arg5=7, arg6=8)
```

```
fn(3, 5, 2, 4, 7, 8)
fn(arg1=3, arg2=5, arg3=2, arg4=4)
fn(arg1=3, arg2=5, arg3=2, arg4=4, arg5=7, arg6=8)
```

Handle Every Situation

```
def fn(*args, **kwargs):
    print(args)  # a tuple of positional arguments
    print(kwargs)  # a dictionary of keyword arguments
```

Function Attributes

PEP 232

Functions already have a number of attributes such as `__doc__`, `__annotations__`, `__defaults__`, etc. Like everything in Python, functions are also objects, therefore user can add a dictionary as attributes by using get / set methods to `__dict__`.

setattr(*object*, *name*, *value*)

This is the counterpart of `getattr()`. The arguments are an object, a string, and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

name need not be a Python identifier as defined in Identifiers and keywords unless the object chooses to enforce that, for example in a custom `__getattribute__()` or via `__slots__`. An attribute whose name is not an identifier will not be accessible using the dot notation, but is accessible through `getattr()` etc..

getattr(*object*, *name*)
getattr(*object*, *name*, *default*)

Return the value of the named attribute of *object*. *name* must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, *default* is returned if provided, otherwise `AttributeError` is raised. *name* need not be a Python identifier (see `setattr()`).

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()

float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()

id()
input()

max()
memoryview()
min()

N
next()

O
object()
oct()
open()

print()
property()

set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()

hasattr(*object*, *name*)

The arguments are an object and a string. The result is `True` if the string is the name of one of the object's attributes, `False` if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an `AttributeError` or not.)

D
delattr()

Z

delattr(*object*, *name*)

This is a relative of `setattr()`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, 'foobar')` is equivalent to `del x.foobar`. *name* need not be a Python identifier (see `setattr()`).

Nested Scopes

PEP 227

Like attributes, function objects can also have methods. These methods can be used as inner functions and can be useful for encapsulation.

```
def parent_function():
    def nested_function():
        print("I'm a nested function.")
    print("I'm a parent function.")
```

```
parent_function()
```

```
parent_function.nested_function()
```

Getter and Setter Methods

```
def point(x, y):
    def set_x(new_x):
        nonlocal x
        x = new_x
    def set_y(new_y):
        nonlocal y
        y = new_y
    def get():
        return x, y
    point.set_x = set_x
    point.set_y = set_y
    point.get = get
    return point
```

Homework for Functions

Week03/functions_firstname_lastname.py

custom_power

- [x] A lambda function
- [x] Two parameters (x and e)
- [x] x is positional-only
- [x] e is positional-or-keyword
- [x] x has the default value 0
- [x] e has the default value 1
- [x] Returns x**e

custom_equation

- [x] A function returns float
- [x] Five integer parameters (x, y, a, b, c)
- [x] x is positional-only with default value 0
- [x] y is positional-only with default value 0
- [x] a is positional-or-keyword with default value 1
- [x] b is positional-or-keyword with default value 1
- [x] c is keyword-only with default value 1
- [x] Function signature must include all annotations
- [x] Docstring must be in reST format.
- [x] Returns (x**a + y**b) / c

fn_w_counter

- [x] A function returns a tuple of an int and a dictionary
- [x] Function must count the number of calls with caller information
- [x] Returning integer is the total number of calls
- [x] Returning dictionary with string keys and integer values includes the caller (_ _name_ _) as key, the number of call coming from this caller as value.

Examples

```
custom_power(2) == 2
custom_power(2, 3) == 8
custom_power(2, e=2) == 4
custom_equation(2, 3) == 5.0
custom_equation(2, 3, 2) == 7.0
custom_equation(2, 3, 2, 3) == 31.0
custom_equation(3, 5, a=2, b=3, c=4) == 33.5
custom_equation(3, 5, 2, b=3, c=4) == 33.5
custom_equation(3, 5, 2, 3, c=4) == 33.5
for i in range(10):
    fn_w_counter()
fn_w_counter() == (11, {'__main__': 11})
```