Parallel Programming

Assoc. Prof. Dr. Bora Canbula

https://github.com/canbula/ParallelProgramming/

Data Structures in Python
Functions and Decorators in Python
Coroutines and Concurrency with **`asyncio`**
IO-bound Problems and Concurrency
Creating Threads in Python with **`threading`**
Global Interpreter Lock and JIT Compiler
Protecting Resources with Lock
Deadlock and Semaphore
Barriers and Conditions
Creating Processes with **`multiprocessing`**
Pipes and Queues
CPU-bound Problems and Parallelism
Creating Clusters
Load Balancing with Containers

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.

Rules

- 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.

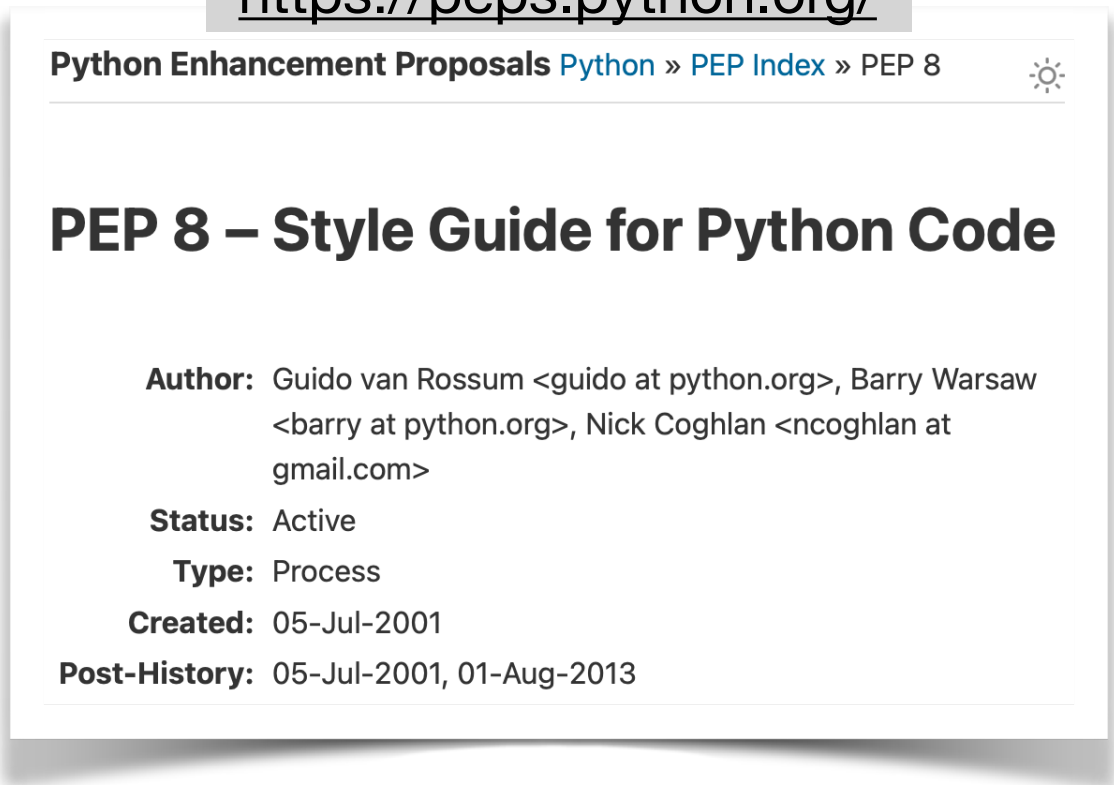
Identifier Names

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

Rules

- 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.

https://peps.python.org/



Python Enhancement Proposals Python » PEP Index » PEP 8

PEP 8 – Style Guide for Python Code

Author: Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>
Status: Active
Type: Process
Created: 05-Jul-2001
Post-History: 05-Jul-2001, 01-Aug-2013

Identifier Names

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

Conventions

- Names to Avoid
  Never use the characters ‘l’ (lowercase letter el), ‘O’ (uppercase letter oh), or ‘I’ (uppercase letter eye) as single character variable names.
- Packages
  Short, all-lowercase names without underscores
- Modules
  Short, all-lowercase names, can have underscores
- Classes
  CapWords (upper camel case) convention
- Functions
  snake_case convention
- Variables
  snake_case convention
- Constants
  ALL_UPPERCASE, words separated by underscores

Leading and Trailing Underscores

- `_single_leading_underscore`
  Weak “internal use” indicator.
  `from M import *` does not import objects whose names start with an underscore.
- `single_trailing_underscore_`
  Used by convention to avoid conflicts with keyword.
- `__double_leading_underscore`
  When naming a class attribute, invokes name mangling (inside class `FooBar`, `__boo` becomes `_FooBar__boo`)
- `__double_leading_and_trailing_underscore__`
  “magic” objects or attributes that live in user-controlled namespaces (`__init__`, `__import__`, etc.). Never invent such names; only use them as documented.

Variable Types

Python is dynamically typed. Python does not have primitive types. Everything is an object in Python, therefore, a variable is purely a reference to an object with the specified value.

Numeric Types

- Integer
- Float
- Complex
- Boolean

Formatted Output

```
print(“static text = ”, variable)
print(“static text = %d” % (variable))
print(“static text = {0}”.format(variable))
print(f”static text = {variable}”)
print(f”static text = {variable:5d}”)
```

Variable Types

Python is dynamically typed. Python does not have primitive types. Everything is an object in Python, therefore, a variable is purely a reference to an object with the specified value.

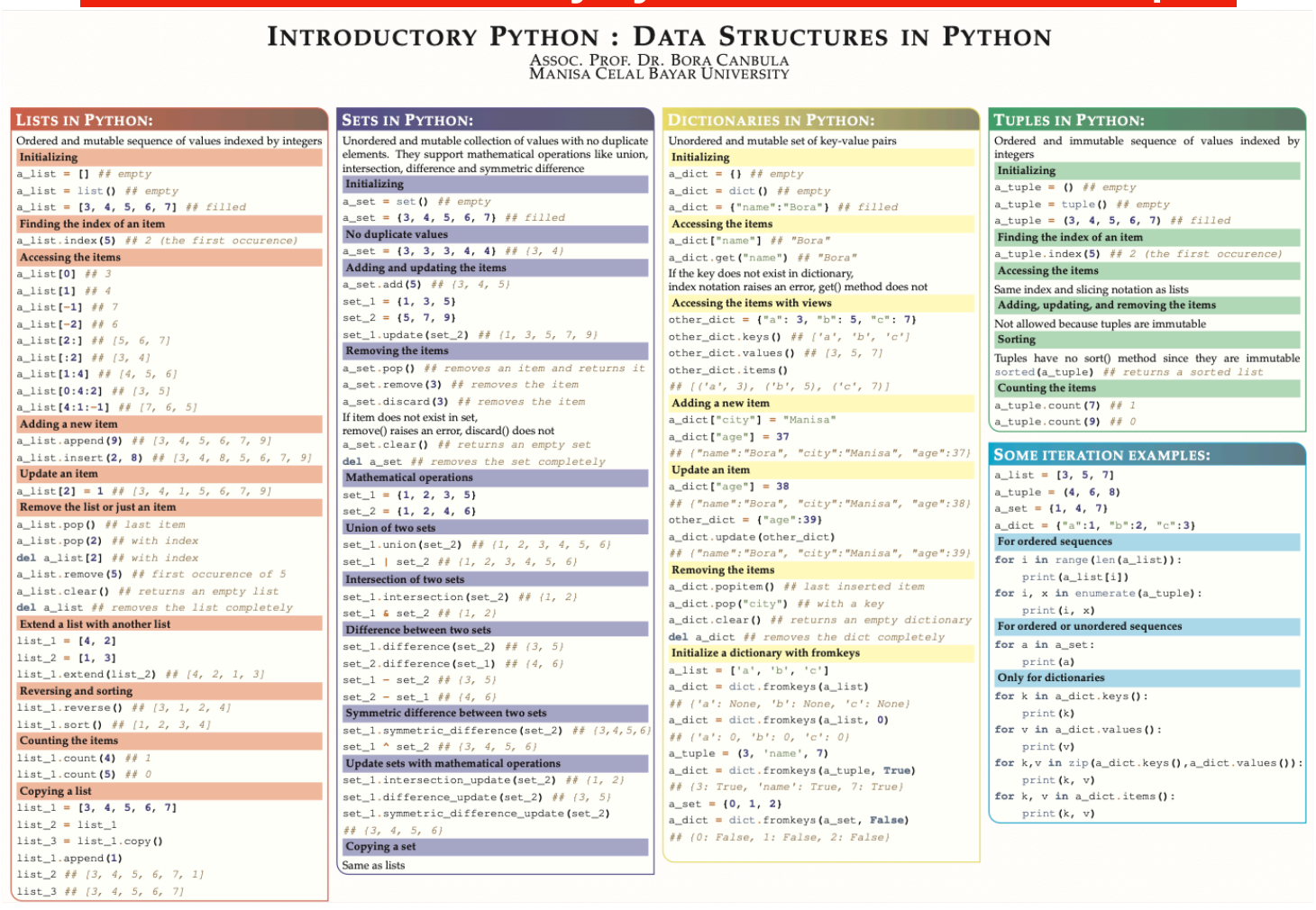
Numeric Types

- Integer
- Float
- Complex
- Boolean

Sequences

- Strings
- List
- Tuple
- Set
- Dictionary

Week02/IntroductoryPythonDataStructures.pdf



INTRODUCTORY PYTHON : DATA STRUCTURES IN PYTHON

ASSOC. PROF. DR. BORA CANBULA
MANISA CELAL BAYAR UNIVERSITY

LISTS IN PYTHON:

Ordered and mutable sequence of values indexed by integers

Initializing
```
a_list = [] ## empty
a_list = list() ## empty
a_list = [3, 4, 5, 6, 7] ## filled
```
Finding the index of an item
```
a_list.index(5) ## 2 (the first occurence)
```
Accessing the items
```
a_list[0] ## 3
a_list[1] ## 4
a_list[-1] ## 7
a_list[-2] ## 6
a_list[2:] ## [5, 6, 7]
a_list[:2] ## [3, 4]
a_list[1:4] ## [4, 5, 6]
a_list[0:4:2] ## [3, 5]
a_list[4:1:-1] ## [7, 6, 5]
```
Adding a new item
```
a_list.append(9) ## [3, 4, 5, 6, 7, 9]
a_list.insert(2, 8) ## [3, 4, 8, 5, 6, 7, 9]
```
Update an item
```
a_list[2] = 1 ## [3, 4, 1, 5, 6, 7, 9]
```
Remove the list or just an item
```
a_list.pop() ## last item
a_list.pop(2) ## with index
del a_list[2] ## with index
a_list.remove(5) ## first occurence of 5
a_list.clear() ## returns an empty list
del a_list ## removes the list completely
```
Extend a list with another list
```
list_1 = [4, 2]
list_2 = [1, 3]
list_1.extend(list_2) ## [4, 2, 1, 3]
```
Reversing and sorting
```
list_1.reverse() ## [3, 1, 2, 4]
list_1.sort() ## [1, 2, 3, 4]
```
Counting the items
```
list_1.count(4) ## 1
list_1.count(5) ## 0
```
Copying a list
```
list_1 = [3, 4, 5, 6, 7]
list_2 = list_1
list_3 = list_1.copy()
list_1.append(1)
list_2 ## [3, 4, 5, 6, 7, 1]
list_3 ## [3, 4, 5, 6, 7]
```

SETS IN PYTHON:

Unordered and mutable collection of values with no duplicate elements. They support mathematical operations like union, intersection, difference and symmetric difference

Initializing
```
a_set = set() ## empty
a_set = {3, 4, 5, 6, 7} ## filled
```
No duplicate values
```
a_set = {3, 3, 3, 4, 4} ## {3, 4}
```
Adding and updating the items
```
a_set.add(5) ## {3, 4, 5}
set_1 = {1, 3, 5}
set_2 = {5, 7, 9}
set_1.update(set_2) ## {1, 3, 5, 7, 9}
```
Removing the items
```
a_set.pop() ## removes an item and returns it
a_set.remove(3) ## removes the item
a_set.discard(3) ## removes the item
```
If item does not exist in set,
remove() raises an error, discard() does not
```
a_set.clear() ## returns an empty set
del a_set ## removes the set completely
```
Mathematical operations
```
set_1 = {1, 2, 3, 5}
set_2 = {1, 2, 4, 6}
```
Union of two sets
```
set_1.union(set_2) ## {1, 2, 3, 4, 5, 6}
set_1 | set_2 ## {1, 2, 3, 4, 5, 6}
```
Intersection of two sets
```
set_1.intersection(set_2) ## {1, 2}
set_1 & set_2 ## {1, 2}
```
Difference between two sets
```
set_1.difference(set_2) ## {3, 5}
set_2.difference(set_1) ## {4, 6}
set_1 - set_2 ## {3, 5}
set_2 - set_1 ## {4, 6}
```
Symmetric difference between two sets
```
set_1.symmetric_difference(set_2) ## {3,4,5,6}
set_1 ^ set_2 ## {3, 4, 5, 6}
```
Update sets with mathematical operations
```
set_1.intersection_update(set_2) ## {1, 2}
set_1.difference_update(set_2) ## {3, 5}
set_1.symmetric_difference_update(set_2)
## {3, 4, 5, 6}
```
Copying a set

Same as lists

DICTIONARIES IN PYTHON:

Unordered and mutable set of key-value pairs

Initializing
```
a_dict = {} ## empty
a_dict = dict() ## empty
a_dict = {"name":"Bora"} ## filled
```
Accessing the items
```
a_dict["name"] ## "Bora"
a_dict.get("name") ## "Bora"
```
If the key does not exist in dictionary,
index notation raises an error, get() method does not

Accessing the items with views
```
other_dict = {"a": 3, "b": 5, "c": 7}
other_dict.keys() ## ['a', 'b', 'c']
other_dict.values() ## [3, 5, 7]
other_dict.items()
## [('a', 3), ('b', 5), ('c', 7)]
```
Adding a new item
```
a_dict["city"] = "Manisa"
a_dict["age"] = 37
## {"name":"Bora", "city":"Manisa", "age":37}
```
Update an item
```
a_dict["age"] = 38
## {"name":"Bora", "city":"Manisa", "age":38}
other_dict = {"age":39}
a_dict.update(other_dict)
## {"name":"Bora", "city":"Manisa", "age":39}
```
Removing the items
```
a_dict.popitem() ## last inserted item
a_dict.pop("city") ## with a key
a_dict.clear() ## returns an empty dictionary
del a_dict ## removes the dict completely
```
Initialize a dictionary with fromkeys
```
a_list = ['a', 'b', 'c']
a_dict = dict.fromkeys(a_list)
## {'a': None, 'b': None, 'c': None}
a_dict = dict.fromkeys(a_list, 0)
## {'a': 0, 'b': 0, 'c': 0}
a_tuple = (3, 'name', 7)
a_dict = dict.fromkeys(a_tuple, True)
## {3: True, 'name': True, 7: True}
a_set = {0, 1, 2}
a_dict = dict.fromkeys(a_set, False)
## {0: False, 1: False, 2: False}
```

TUPLES IN PYTHON:

Ordered and immutable sequence of values indexed by integers

Initializing
```
a_tuple = () ## empty
a_tuple = tuple() ## empty
a_tuple = (3, 4, 5, 6, 7) ## filled
```
Finding the index of an item
```
a_tuple.index(5) ## 2 (the first occurence)
```
Accessing the items

Same index and slicing notation as lists

Adding, updating, and removing the items

Not allowed because tuples are immutable

Sorting

Tuples have no sort() method since they are immutable
```
sorted(a_tuple) ## returns a sorted list
```
Counting the items
```
a_tuple.count(7) ## 1
a_tuple.count(9) ## 0
```

SOME ITERATION EXAMPLES:
```
a_list = [3, 5, 7]
a_tuple = (4, 6, 8)
a_set = {1, 4, 7}
a_dict = {"a":1, "b":2, "c":3}
```
For ordered sequences
```
for i in range(len(a_list)):
    print(a_list[i])
for i, x in enumerate(a_tuple):
    print(i, x)
```
For ordered or unordered sequences
```
for a in a_set:
    print(a)
```
Only for dictionaries
```
for k in a_dict.keys():
    print(k)
for v in a_dict.values():
    print(v)
for k,v in zip(a_dict.keys(),a_dict.values()):
    print(k, v)
for k, v in a_dict.items():
    print(k, v)
```

Variable Types

Python is dynamically typed. Python does not have primitive types. Everything is an object in Python, therefore, a variable is purely a reference to an object with the specified value.

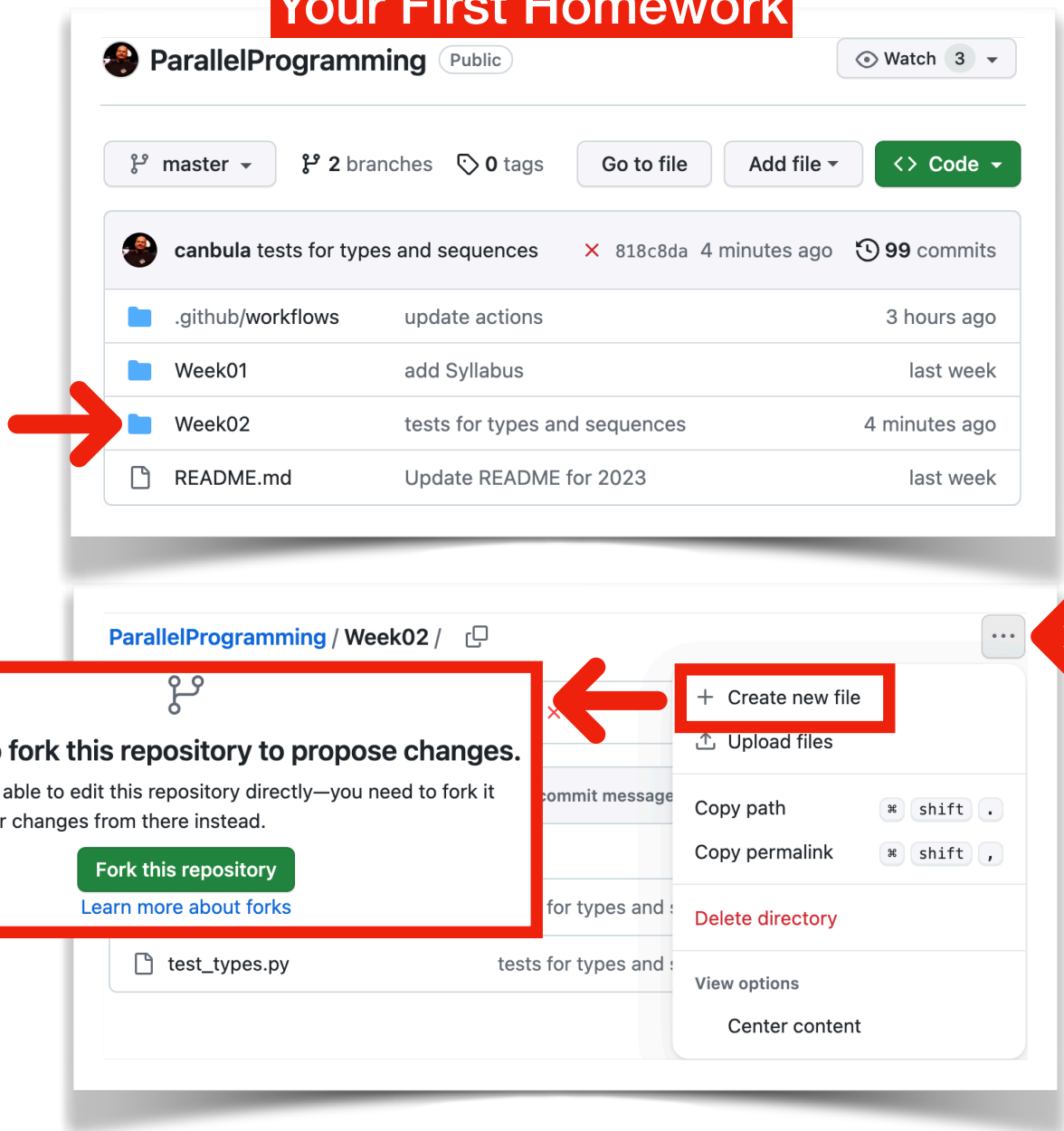
Numeric Types

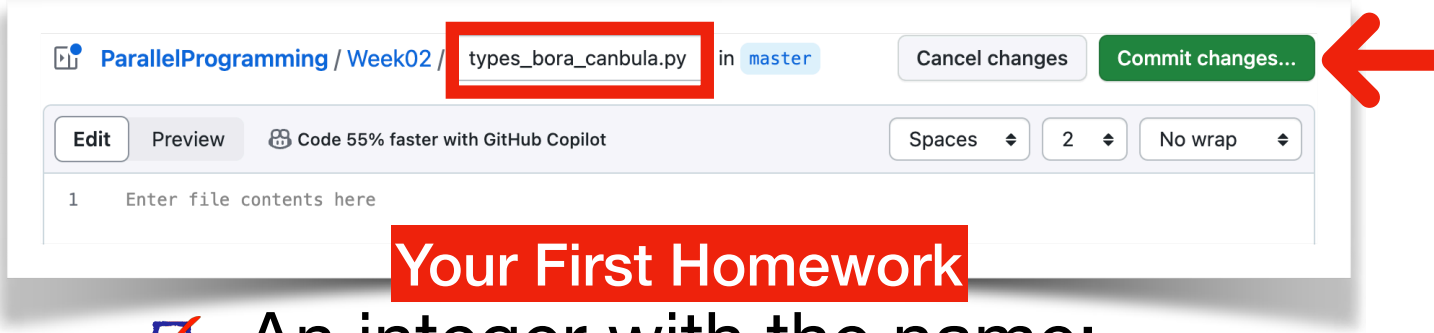
- Integer
- Float
- Complex
- Boolean

Sequences

- Strings
- List
- Tuple
- Set
- Dictionary

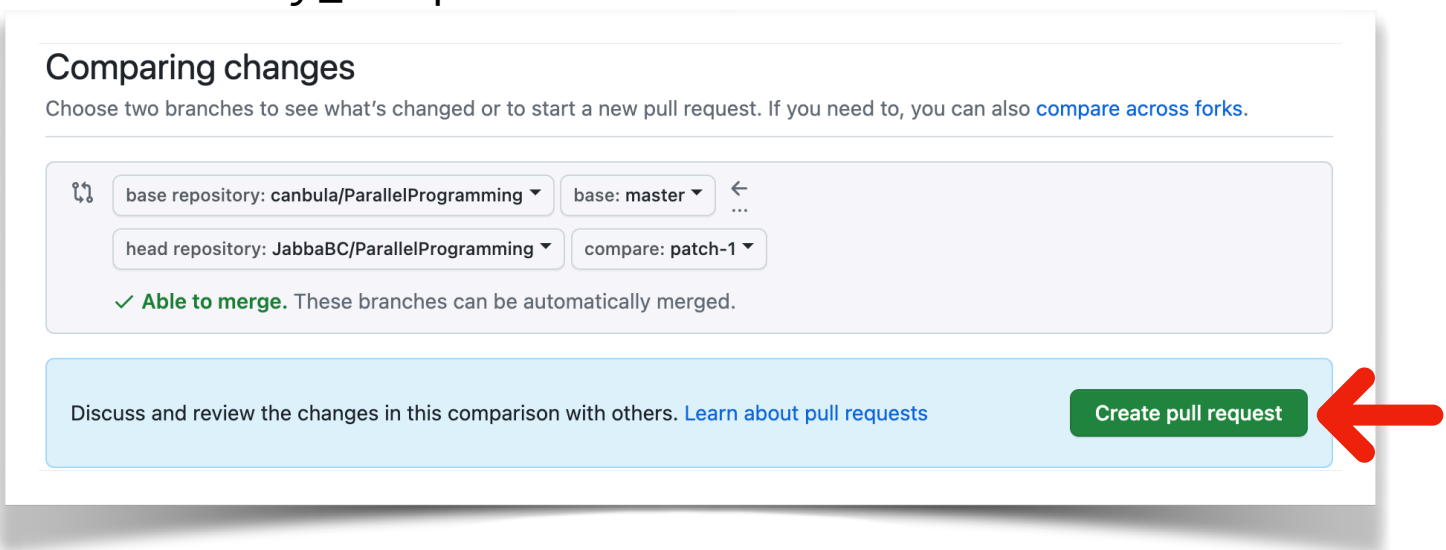
Your First Homework

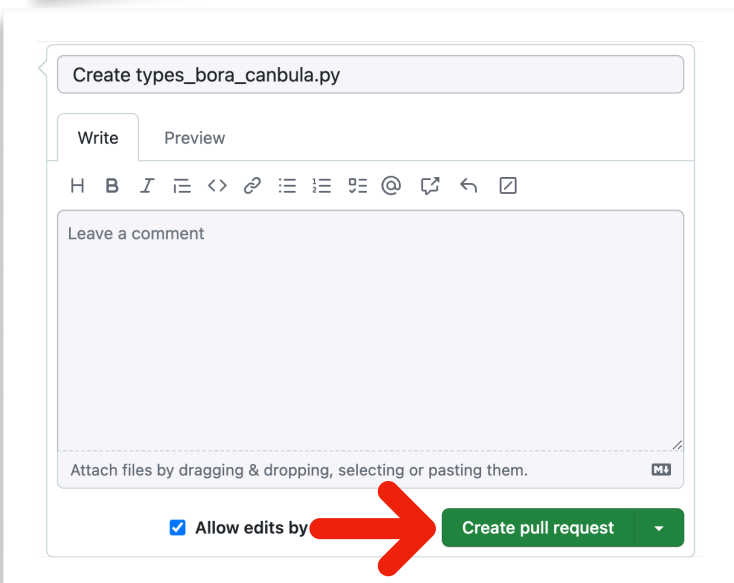


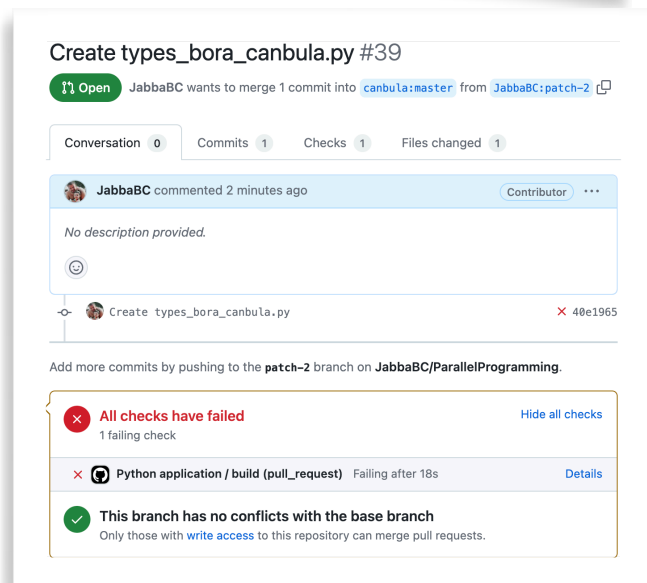


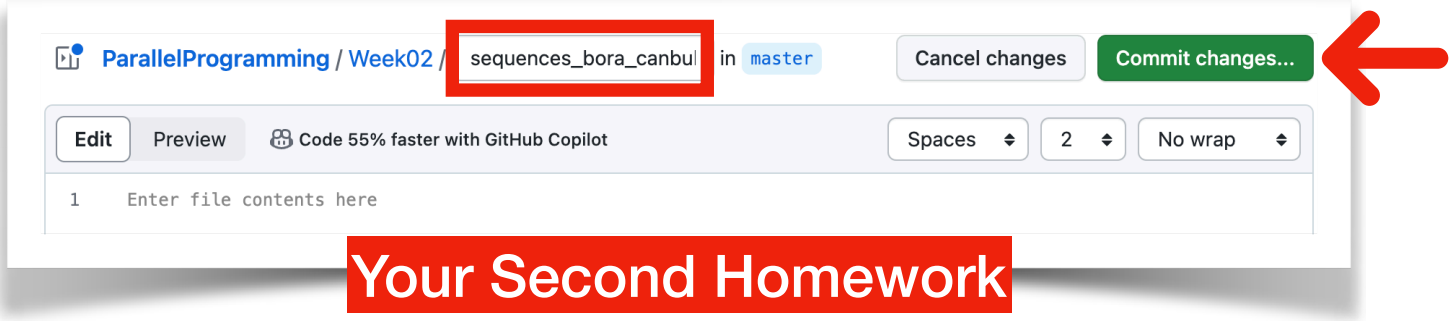
Your First Homework

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









Your Second Homework

- A list with the name:
  `my_list`
- A tuple with the name:
  `my_tuple`
- A set with the name:
  `my_set`
- A dictionary with the name:
  `my_dict`
- A function with the name:
  `remove_duplicates (list -> list)`
  to remove duplicate items from a list
- A function with the name:
  `list_counts (list -> dict)`
  to count the occurrence of each item in a list and return as a dictionary
- A function with the name:
  `reverse_dict (dict -> dict)`
  to reverse a dictionary, switch values and keys with each other.

Problem Set

1. What is the correct writing of the programming language that we used in this course?

( ) Phyton
( ) Pyhton
( ) Pthyon
( ) Python

2. What is the output of the code below?

```
my_name = "Bora Canbula"
print(my_name[2::-1])
```

( ) alu
( ) ula
( ) roB
( ) Bor

3. Which one is not a valid variable name?

( ) for_
( ) Manisa_Celal_Bayar_University
( ) IF
( ) not

4. What is the output of the code below?

```
for i in range(1, 5):
  print(f"{i:2d}{(i/2):4.2f}", end='')
```

( ) 010.50021.00031.50042.00
( ) 10.50 21.00 31.50 42.00
( ) 1 0.5 2 1.0 3 1.5 4 2.0
( ) 100.5 201.0 301.5 402.0

5. Which one is the correct way to print Bora's age?

```
profs = [
  {"name": "Yener", "age": 25},
  {"name": "Bora", "age": 37},
  {"name": "Ali", "age": 42}
]
```

( ) profs["Bora"]["age"]
( ) profs[1][1]
( ) profs[1]["age"]
( ) profs.age[name="Bora"]

6. What is the output of the code below?

```
x = set([int(i/2) for i in range(8)])
print(x)
```

( ) {0, 1, 2, 3, 4, 5, 6, 7}
( ) {0, 1, 2, 3}
( ) {0, 0, 1, 1, 2, 2, 3, 3}
( ) {0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4}

7. What is the output of the code below?

```
x = set(i for i in range(0, 4, 2))
y = set(i for i in range(1, 5, 2))
print(x^y)
```

( ) {0, 1, 2, 3}
( ) {}
( ) {0, 8}
( ) SyntaxError: invalid syntax

8. Which of the following sequences is immutable?

( ) List
( ) Set
( ) Dictionary
( ) String

9. What is the output of the code below?

```
print(int(2_999_999.999))
```

( ) 2
( ) 3000000
( ) ValueError: invalid literal
( ) 2999999

10. What is the output of the code below?

```
x = (1, 5, 1)
print(x, type(x))
```

( ) [1, 2, 3, 4] <class 'list'>
( ) (1, 5, 1) <class 'range'>
( ) (1, 5, 1) <class 'tuple'>
( ) (1, 2, 3, 4) <class 'set'>

Parallel Programming

Assoc. Prof. Dr. Bora Canbula

https://github.com/canbula/ParallelProgramming/

Functions

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

Naming Convention from PEP8

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

Basic Function Definition

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

Input and 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 (> Python 3.10)

```
def fn(arg1: int|float, arg2: int|float) -> (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 number."""
    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 number."""
    return arg1 + arg2
```

Multi-line Docstrings

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

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

Docutils and Sphinx are tools to automatically create documentations

reST (reStructuredText)

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

    :param arg1: First number
    :param arg2: Second number
    :return: Sum of two numbers
    """
    return arg1 + arg2
```

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

Google

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

    Args:
        arg1 (int): First number
        arg2 (int): Second number

    Returns:
        int: Sum of two numbers
    """
    return arg1 + arg2
```

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.

Positional-or-Keyword (Standard Binding)

```
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 and Keyword-Only

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

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

Positional-Only and Positional-or-Keyword and Keyword-Only

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

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

PEP 457

*args and **kwargs

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

fn(), fn(3), fn(3, 5), fn(x=3, y=5), fn(3, 5, x=2, y=4),
fn(3, 5, x=2, y=4, 1, 2)
```

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__.

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
	float()	max()	set()
B	format()	memoryview()	setattr()
bin()	frozenset()	min()	slice()
bool()			sorted()
breakpoint()	G	N	staticmethod()
bytearray()	getattr()	next()	str()
bytes()	globals()		sum()
		O	super()
C	H	object()	
callable()	hasattr()	oct()	T
chr()	hash()	open()	tuple()
classmethod()	help()	ord()	type()
compile()	hex()		
complex()		P	V
	I	pow()	vars()
D	id()	print()	
delattr()	input()	property()	Z
dict()	int()		zip()
dir()	isinstance()		
divmod()	issubclass()		_
	iter()		__import__()

Function Attributes PEP 232

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()`).

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.)

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("Nested function")
    print("Parent function")
    parent_function.nested_function = nested_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
```

Decorators

Decorators take a function as argument and returns a function. They are used to extend the behavior of the wrapped function, without modifying it. So they are very useful for dealing with code legacy.

Traditional Way

```
def my_decorator(fn):
    def _my_decorator():
        print("Before function")
        fn()
        print("After function")
    return _my_decorator

def my_decorated_function():
    print("Function")

my_decorated_function = \
    my_decorator(my_decorated_function)
```

Pythonic Way

```
def d1(fn):
    def _d1():
        print("Before d1")
        fn()
        print("After d1")
    return _d1

@d1
def f1():
    print("Function")
```

Decorators with Arguments

```
def decorator(func):
    def _decorator(*args, **kwargs):
        print("I am decorator")
        print(args)
        print(kwargs)
        func(*args, **kwargs)
    return _decorator

@decorator
def decorated_func_w_args(x):
    print(f"x = {x}")

@decorator
def decorated_triple_print(
    x=None, y=None, z=None):
    x_str = f"x = {x} " \
        if x is not None else f""
    y_str = f"y = {y} " \
        if y is not None else f""
    z_str = f"z = {z} " \
        if z is not None else f""
    print(x_str + y_str + z_str)
```

Decorator Chain

```
def d1(func):
    def _d1(*args, **kwargs):
        print(f"d1 here for \
              {func.__name__}")
        func(*args, **kwargs)
    return _d1

def d2(func):
    def _d2(*args, **kwargs):
        print(f"d2 here for \
              {func.__name__}")
        func(*args, **kwargs)
    return _d2

@d1
@d2
def fd(x):
    print(f"f says {x}")
```

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

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})
```

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.

Homework for Decorators

Week03/decorators_firstname_lastname.py

performance

- A decorator which measures the performance of functions and also saves some statistics.
- Has three attributes: counter, total_time, total_mem
- Attribute counter stores the number of times that the decorator has been called.
- Attribute total_time stores the number of total time that the functions took.
- Attribute total_mem stores the total memory in bytes that the functions consumed.

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.

Problem Set

1. Does a Python function always return a value?

( ) True
( ) False

2. Which of the following is the valid start to define a function in Python?

( ) define func():
( ) function func() {
( ) void func():
( ) def func():

3. What does return from call mltpl(2,3)?

```
def mltpl(a, b=1):
    return a*b
```

Your Answer:

4. How can you use the following function to print exactly 'ParallelProgramming'?

```
def a(x):
    def b(y):
        print(y, end='')
    print(x, end='')
```

( ) a('Parallel Programming')
( ) a('Parallel');b('Programming')
( ) a('Parallel');a('Programming')
( ) a.b('ParallelProgramming')

5. How can you change 'BC' with your own initials in the following function?

```
def speak(s):
    if not speak.who:
        speak.who = 'BC'
    print(f"{speak.who} says {s}")
```

Your Answer:

6. There is a module called 'logging' to employ logging facility in Python.

```
import logging
logging.info('Just a normal message')
logging.warning('Not fatal but still noted')
logging.error('There is something wrong')
```

You are expected to implement logging feature to an existing code which uses the function below.

```
def my_ugly_debug(s, level=0):
    pre_text = [
        "INFO",
        "WARNING",
        "ERROR"
    ]
    print(f"{pre_text[level]}: {s}")
```

You are not allowed to make changes in my_ugly_debug, so find another way.

Parallel Programming

Assoc. Prof. Dr. Bora Canbula

https://github.com/canbula/ParallelProgramming/

Problems Types

One can categorize the problems in computer programming based on the primary source of their performance bottlenecks.

I/O-bound Problems

While solving an **I/O-bound** problem, the system spends a significant amount of time waiting for input/output operations.

Subcategories can be **Disk I/O** (reading or writing to a hard drive) and **Network I/O** (waiting for data from a remote server).

The solutions often involve **asynchronous programming**, caching, or optimizing the I/O operations.

CPU-bound Problems

For **CPU-bound** problems, computational processing is the bottleneck.

Speeding up the computation requires either a faster CPU or optimizing the computation itself.

Parallel processing, **algorithm optimization**, or offloading computations to other systems or specialized hardware (like **GPUs**) are common strategies to overcome these problems.

Memory-bound Problems

Problems where the primary constraint is the **system's memory**.

Solutions can involve **optimizing data structures**, utilizing **external memory storage**, or employing algorithms that are more **memory-efficient**.

Coroutines

Coroutines are a generalization of subroutines (functions) used for **cooper** that return once and c**pause** their execution and later continue from where they left off.

Do you know any Callable which can pause its execution?

return

Regular functions returning a specified value back to the caller and terminates the function's execution.

Once the function returns a value using `return`, **its state lost**. Subsequent calls to the function start the execution from the beginning of the function.

Used to compute a value and return it to caller immediately.

yield

Used in special functions known as **generators**. Produces a series of values for iteration using a **lazy evaluation** approach (values are generated on-the-fly, not stored in memory).

When a function using the `yield` keyword is called, it returns a **generator** object without even beginning execution of the function.

Upon calling `next()`, the function runs until it encounters the `yield` keyword. The function's execution is **paused**, and the yielded value is returned. Subsequent calls to `next()` resume the function's execution immediately after the last `yield` statement.

Once all values have been yielded, the generator raises a `StopIteration` exception.

Can you write your own generator by using some magic methods in a class?

Coroutines

Coroutines are a generalization of subroutines (functions) used for **cooperative multitasking**. Unlike functions that return once and do not maintain state, **coroutines can pause** their execution and later continue from where they left off.

Traditional synchronous programming runs line by li

In I/O-bound problems, the program waits for the opera

Asynchronous programming allows tasks to run concurrently.

Increasing Efficiency

Can you increase the efficiency in real world problems with this technique?

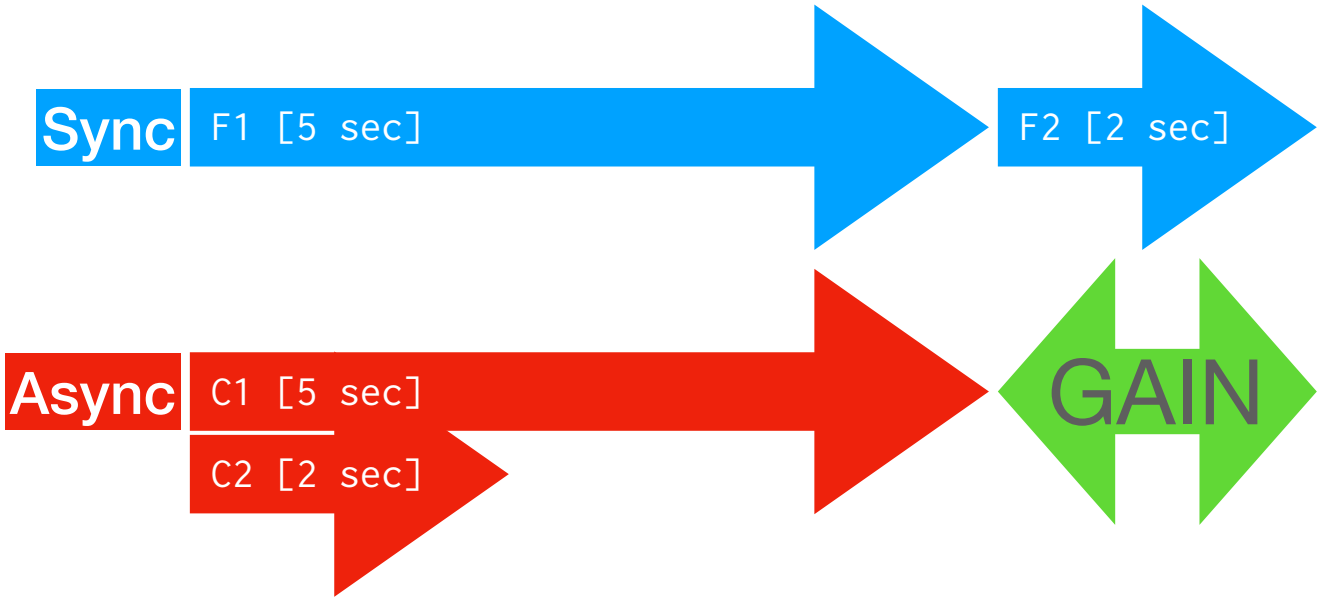
https://youtu.be/loqCY9b7aec
https://www.canbula.com/cookie

How to implement Coroutines in Python?

Coroutines declared with the **`async/await`** syntax is the best practice of writing asynchronous applications in Python.

✓ Handle many tasks concurrently without multi-threading.

✓ Improve performance for I/O-bound tasks.



Can you write your own awaitable by using some magic methods in a class?

Homework for Coroutines

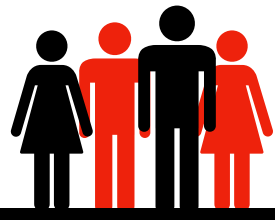
Week04/awaitme_firstname_lastname.py

awaitme

- ☑ A decorator which turns any function into a coroutine.
- ☑ It must pass all the arguments to the function properly.
- ☑ If function returns any value, so the decorator returns it.

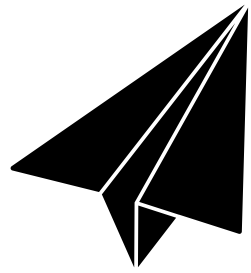
Rules for your pull requests

- ✓ 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.



PROJECT: Implement the cookie problem in Python

- ✓ Application optimizes any recipe.
- ✓ You can group up to 4 students.
- ✓ Backend with Python.
- ✓ Tests with Pytest.
- ✓ Frontend with HTML + CSS + JS.
- ✓ 5 minutes presentation in English.



Problem Set

1. Write the sum_of_digits function which satisfies the tests given below.

```python
def sum_of_digits(n: int) -> int:
    pass


if __name__ == "__main__":
    # tests for integer values
    tests = [[1, 1], [23, 5], [1001, 2], [5623, 16]]
    for x in tests:
        if not sum_of_digits(x[0]) == x[1]:
            str = f"Value test is failed for {x[0]}"
            exit(str)
    # tests for non-integer values
    tests = [1.5, 1 + 2j, "a", True]
    for x in tests:
        if not sum_of_digits(x) == TypeError:
            str = f"Type test is failed for {x}"
            exit(str)
    print("Tests are completed.")
```

2. Rewrite the test part of the code given in question **1** by using logging module.

3. The url www.canbula.com/prime/{n} returns a dictionary including the prime numbers below an integer n.
Example:
  Request: https://www.canbula.com/prime/5
  Response: {"n":"5","primes":[2,3]}

We want to test this service but the problem is response times are really long. Therefore you are requested to:

- Write tests for almost all scenerios
- Your tests should be running asynchronously so we don't have to wait for every test sequentially
- If you still have some extra time, develop your own project with Flask or FastAPI, which satisfies your tests.