PARALLEL PROGRAMMING

Assoc. Prof. Dr. Bora Canbula

Manisa Celal Bayar University
Department of Computer Engineering

https://github.com/canbula/ParallelProgramming/ 302kib9

Parallel 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

Parallel Programming (Teams Code: 302kib9)

We are going to learn the basics of asynchronous programming, creating multiple threads and processes in this course. Python is preferred as the programming language for the applications of this course.

Required Text

Python Concurrency with asyncio, Manning, *Matthew Fowler*

A Practical Approach to High-Performance Computing, Springer, *Sergei Kurgalin – Sergei Borzunov*

Python Parallel Programming Cookbook, Packt, *Giancarlo Zaccone*

Course Materials

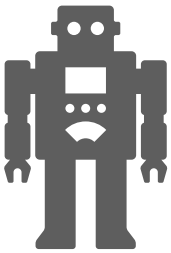
- Python 3.x (Anaconda is preferred)
- Jupyter Notebook from Anaconda
- Pycharm from JetBrains / Microsoft Visual Studio Code
- PC with a Linux distro or a Linux terminal in Windows 10/11.

Course Schedule

Week	Subject	Week	Subject
01	Data Structures in Python	08	Deadlock and Semaphore
02	Functions and Decorators in Python	09	Barriers and Conditions
03	Coroutines and Concurrency with asyncio	10	Creating Processes with multiprocessing
04	IO-bound Problems and Concurrency	11	Pipes and Queues
05	Creating Threads in Python with threading	12	CPU-bound Problems and Parallelism
06	Global Interpreter Lock and JIT Compiler	13	Creating Clusters
07	Protecting Resources with Lock	14	Load Balancing with Containers

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

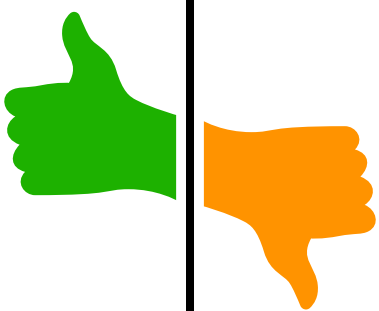
There are two different ways of transforming a program from a high-level programming language into machine language:

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://form[illegible].com/r/3qrM5Gj66X

in-class quizzes

h[illegible]forms.office.com/r/GNNHg5B4c7

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.

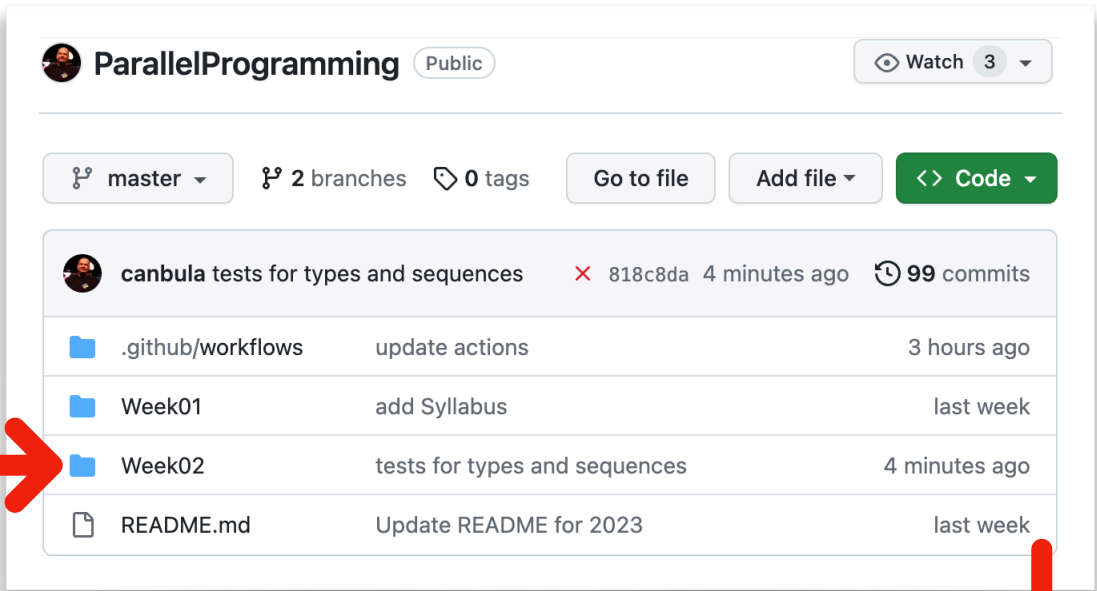
New in version 3.9.

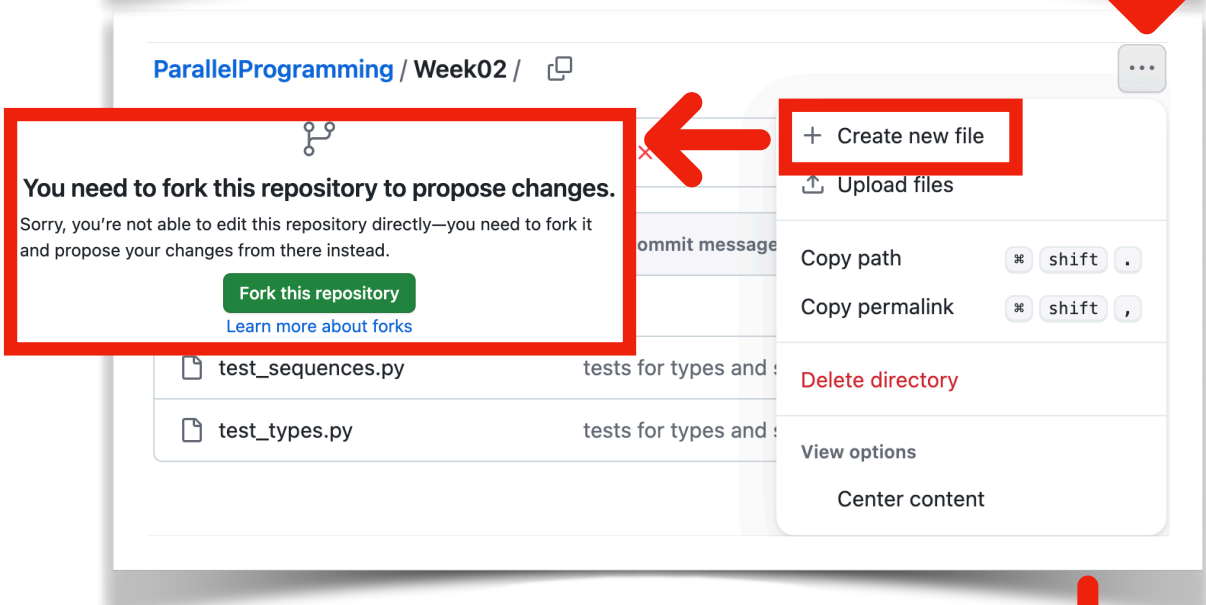
https://form[illegible].com/r/VBi2yJZiX5

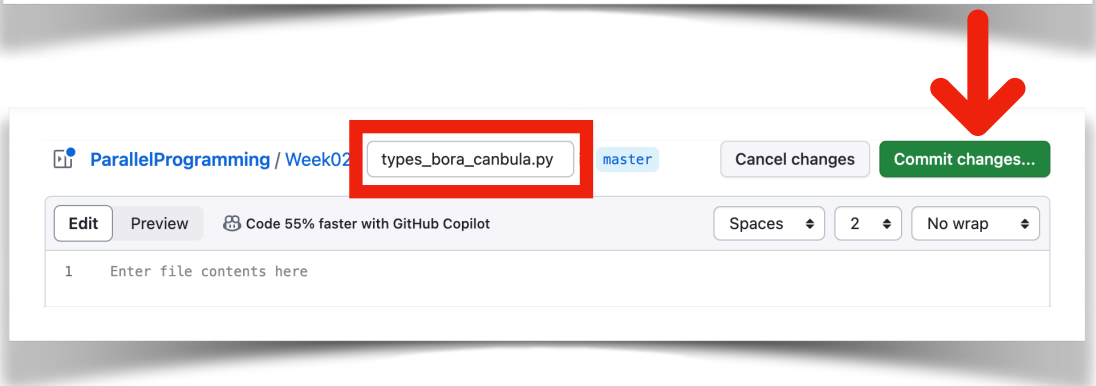
in-class quizzes

h[illegible]forms.office.com/r/RRCyEr8RE8

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

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?

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
```

```
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]
```

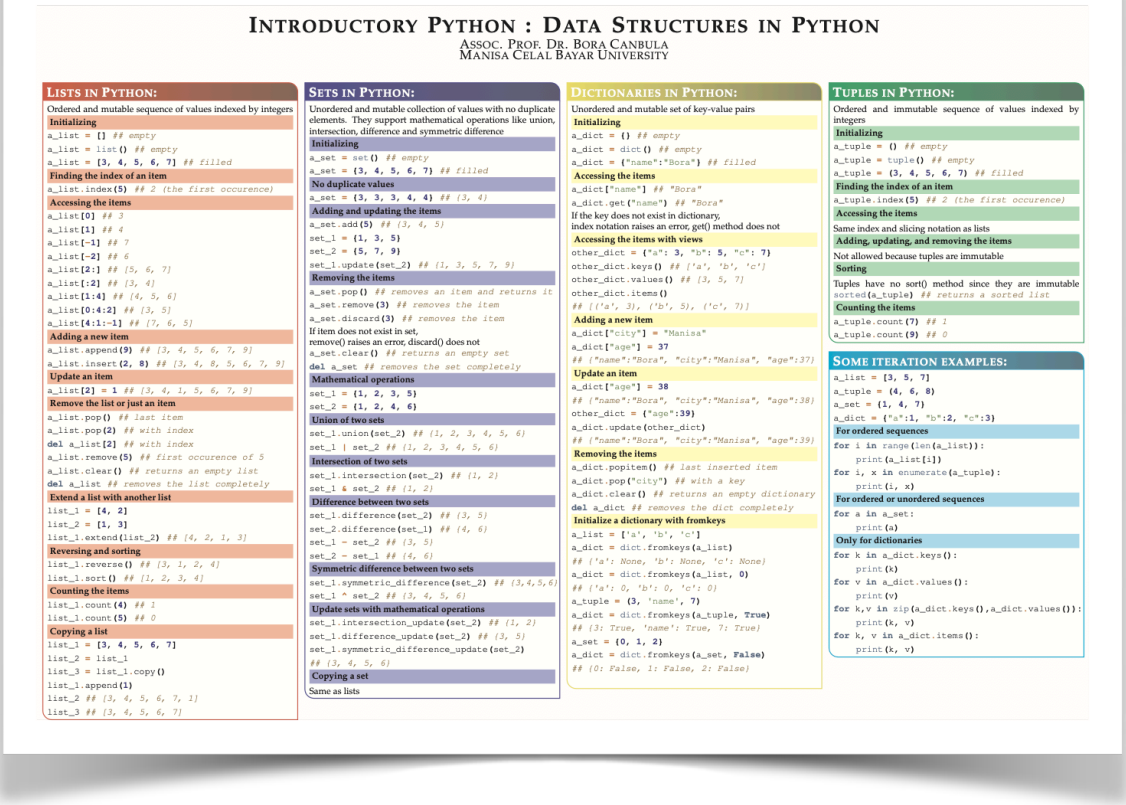
Counting the items

```
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]
```

Week03/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

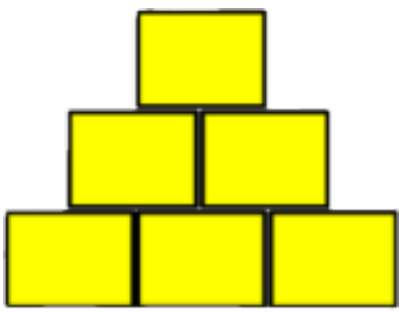
```
for k in a_dict.keys():
    print(k)
for v in a_dict.values():
    print(v)
for k,v in zip(a_dict.keys(),a_dict.values()):
    print(k, v)
for k, v in a_dict.items():
    print(k, v)
```

---

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

https://forms.office.com/r/cpBQGv3NJ0

Week03/pyramid_first_last.py



```python
def calculate_pyramid_height(number_of_blocks):



    return height
```

Week03/sequences_first_last.py

```python
def remove_duplicates(seq: list) -> list:
    """
    This function removes duplicates from a list.
    """
    return ...

def list_counts(seq: list) -> dict:
    """
    This function counts the number of
    occurrences of each item in a list.
    """
    return ...

def reverse_dict(d: dict) -> dict:
    """
    This function reverses the keys
    and values of a dictionary.
    """
    return ...
```

Exception Handling

Exception handling is a mechanism in Python to handle runtime errors gracefully without crashing the program.

```
try:
    # Code that may raise an exception
except Exception as e:
    # Code to handle the exception
else:
    # Code to execute if no exception is raised
finally:
    # Code that will be executed regardless of the result
```

Handling an Exception

```
try:
    1 / 0
except ZeroDivisionError:
    print("You cannot divide by zero.")
```

Handling All Exceptions

```
try:
    number = int(input("Enter a number: "))
    result = 1 / number
except Exception as e:
    print("An error occurred:", e)
```

Using finally Block

```
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found.")
else:
    print("File read successfully.")
    print("Content:", content)
finally:
    file.close()
    print("File closed.")
```

Handling Multiple Exceptions

```
try:
    number = int(input("Enter a number: "))
    result = 1 / number
except ValueError:
    print("Please enter a valid number.")
except ZeroDivisionError:
    print("You cannot divide by zero.")
```

Using else Block

```
try:
    number = int(input("Enter a number: "))
    result = 1 / number
except ValueError:
    print("Please enter a valid number.")
except ZeroDivisionError:
    print("You cannot divide by zero.")
else:
    print("The result is:", result)
```

Raising Exceptions

```
def calculate_area(radius):
    if radius < 0:
        raise ValueError("No negative radius!")
    return 3.14 * radius ** 2

try:
    area = calculate_area(-5)
except ValueError as e:
    print("An error occurred:", e)
```

Can we create custom exceptions?

Identifier Names

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

Naming Conventions from PEP 8

- **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

Underscore Usage in Identifier Names

- **_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.

Functions

Functions are defined by using def keyword, name, and the parenthesized list of formats 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

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()`).

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

Function objects can 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 wrapper 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}")
```

https://forms.office.com/r/kTsvbVSL5G

https://forms.office.com/r/1LvXFpEKVU

in-class quizzes

Week04/functions_firstname_lastname.py

custom_power

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

custom_equation

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

fn_w_counter

- A function returns a tuple of an int and a dictionary
- Function must count the number of calls with caller information
- Returning integer is the total number of calls
- 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})
```

Week04/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.

The problems in computer programming can be categorized 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 are a generalization of subroutines (functions) used for **cooperative** ...once and do not maintain... ...on 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 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.

Increasing Efficiency

Traditional synchronous programming runs line by line.
In I/O-bound problems, the program waits for the operation.
Asynchronous programming allows tasks to run concurrently.

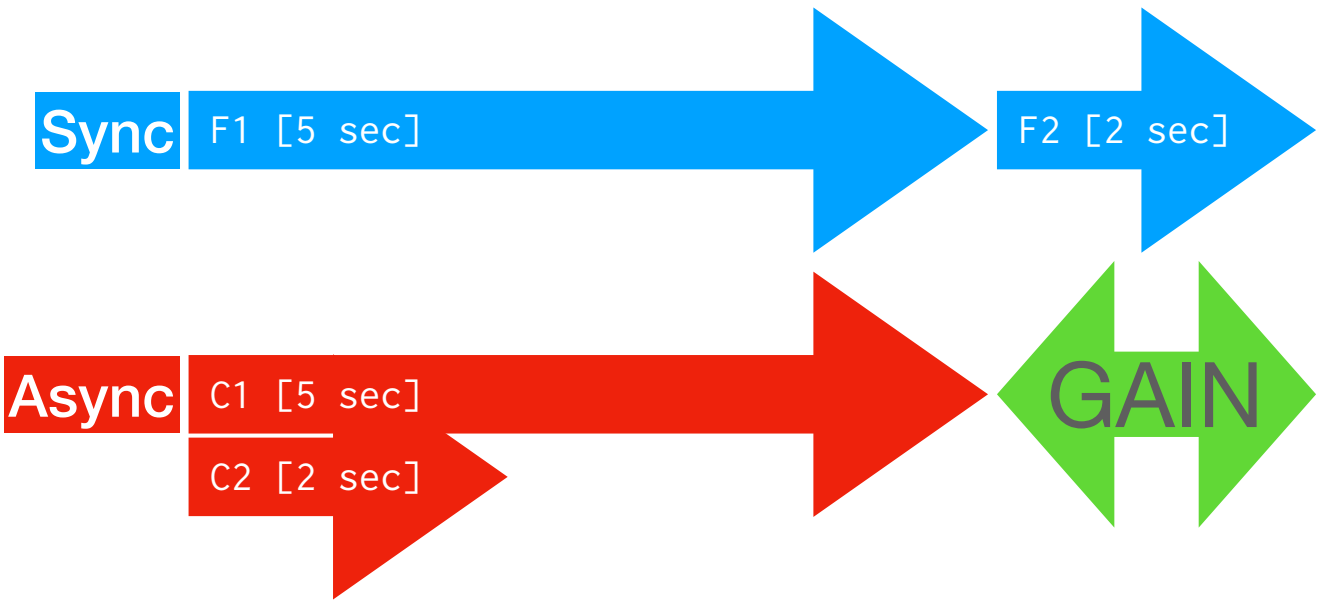
Can you increase the efficiency in real world problems with this technique?

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?

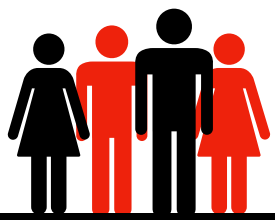
Week05/awaitme_firstname_lastname.py

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.

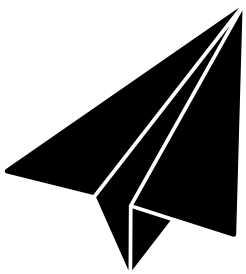
Rules for your pull requests

- ✓ 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.



in-class quiz

https://form[illegible].com/r/NE0XUfgd8E

Wrap-up: Use what we have learned so far in a project!

Using requests module

```
Python 3.12.7 (main, Oct  1 2024, 02:05:46) [Clang 16.0.0 (clang-1600.0.26.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>> dir(requests)
['ConnectTimeout', 'ConnectionError', 'DependencyWarning', 'FileModeWarning', 'HTTPError',
 'JSONDecodeError', 'NullHandler', 'PreparedRequest', 'ReadTimeout', 'Request', 'RequestEx
ception', 'RequestsDependencyWarning', 'Response', 'Session', 'Timeout', 'TooManyRedirects
', 'URLRequired', '__author__', '__author_email__', '__build__', '__builtins__', '__cached
__', '__cake__', '__copyright__', '__description__', '__doc__', '__file__', '__license__',
 '__loader__', '__name__', '__package__', '__path__', '__spec__', '__title__', '__url__',
'__version__', '_check_cryptography', '_internal_utils', 'adapters', 'api', 'auth', 'certs
', 'chardet_version', 'charset_normalizer_version', 'check_compatibility', 'codes', 'compa
t', 'cookies', 'delete', 'exceptions', 'get', 'head', 'hooks', 'logging', 'models', 'optio
ns', 'packages', 'patch', 'post', 'put', 'request', 'session', 'sessions', 'ssl', 'status_
codes', 'structures', 'urllib3', 'utils', 'warnings']
>>> help(requests.get)
```

```
get(url, params=None, **kwargs)
    Sends a GET request.

    :param url: URL for the new
    :param params: (optional) D                                    to send
        in the query string for
    :param \*\*kwargs: Optional
    :return: :class:`Response <
    :rtype: requests.Response
```

```
NAME
    requests.exceptions

DESCRIPTION
    requests.exceptions
    ~~~~~~~~~~~~~~~~~~~

    This module contains the set of Requests' exceptions.

CLASSES
    builtins.OSError(builtins.Exception)
        RequestException
            ChunkedEncodingError
            ConnectionError
                ConnectTimeout(ConnectionError, Timeout)
                ProxyError
                SSLError
            ContentDecodingError(RequestException, urllib3.exceptions.HTTPError)
            HTTPError
            InvalidHeader(RequestException, builtins.ValueError)
            InvalidJSONError
                JSONDecodeError(InvalidJSONError, json.decoder.JSONDecodeError)
            InvalidSchema(RequestException, builtins.ValueError)
            InvalidURL(RequestException, builtins.ValueError)
                InvalidProxyURL
            MissingSchema(RequestException, builtins.ValueError)
            RetryError
            StreamConsumedError(RequestException, builtins.TypeError)
            Timeout
                ReadTimeout
            TooManyRedirects
            URLRequired
            UnrewindableBodyError
    builtins.Warning(builtins.Exception)
        RequestsWarning
            FileModeWarning(RequestsWarning, builtins.DeprecationWarning)
            RequestsDependencyWarning
```

```
>>> r = requests.get("https://www.canb
dir(r)
['__attrs__', '__bool__', '__class__',                                    doc__', '_
_enter__', '__eq__', '__exit__', '__fo                                    getstate__
', '__gt__', '__hash__', '__init__',                                      __lt__',
'__module__', '__ne__', '__new__', '__                                    __repr__
', '__setattr__', '__setstate__', '__s                                    __weakref_
_', '_content', '_content_consumed', '_next', 'apparent_encoding', 'close', 'connection',
'content', 'cookies', 'elapsed', 'encoding', 'headers', 'history', 'is_permanent_redirect'
, 'is_redirect', 'iter_content', 'iter_lines', 'json', 'links', 'next', 'ok', 'raise_for_s
tatus', 'raw', 'reason', 'request', 'status_code', 'text', 'url']
>>> help(r.json)
```

```
json(**kwargs) method of requests.models.Response instance
    Returns the json-encoded content of a response, if any.

    :param \*\*kwargs: Optional arguments that ``json.loads`` takes.
    :raises requests.exceptions.JSONDecodeError: If the response body does not
        contain valid json.
```

Wrap-up: Use what we have learned so far in a project!

Using logging module

```
Python 3.12.7 (main, Oct  1 2024, 02:05:46) [Clang 16.0.0 (clang-1600.0.26.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import logging
>>> print([s for s in dir(logging) if not s.startswith("_")])
['BASIC_FORMAT', 'BufferingFormatter', 'CRITICAL', 'DEBUG', 'ERROR', 'FATAL', 'FileHandler', 'Filter', 'Filterer', 'Formatter', 'GenericAlias', 'Handler', 'INFO', 'LogRecord', 'Logger', 'LoggerAdapter', 'Manager', 'NOTSET', 'NullHandler', 'PercentStyle', 'PlaceHolder', 'RootLogger', 'StrFormatStyle', 'StreamHandler', 'StringTemplateStyle', 'Template', 'WARN', 'WARNING', 'addLevelName', 'atexit', 'basicConfig', 'captureWarnings', 'collections', 'critical', 'currentframe', 'debug', 'disable', 'error', 'exception', 'fatal', 'getHandlerByName', 'getHandlerNames', 'getLevelName', 'getLevelNamesMapping', 'getLogRecordFactory', 'getLogger', 'getLoggerClass', 'info', 'io', 'lastResort', 'log', 'logAsyncioTasks', 'logMultiprocessing', 'logProcesses', 'logThreads', 'makeLogRecord', 'os', 'raiseExceptions', 're', 'root', 'setLogRecordFactory', 'setLoggerClass', 'shutdown', 'sys', 'threading', 'time', 'traceback', 'warn', 'warning', 'warnings', 'weakref']
```

```
debug(msg, *args, **kwargs)
    Log a message with severity 'DEBUG' on the root logger. If the logger has
    no handlers, call basicConfig() to add a console handler with a pre-defined
    format.
```

```
basicConfig(**kwargs)
    Do basic configuration for the logging system.

    This function does nothing if the root logger already has handlers
    configured, unless the keyword argument *force* is set to ``True``.
    It is a convenience method intended for use by simple scripts
    to do one-shot configuration of the logging package.

    The default behaviour is to create a StreamHandler which writes to
    sys.stderr, set a formatter using the BASIC_FORMAT format string, and
    add the handler to the root logger.

    A number of optional keyword arguments may be specified, which can alter
    the default behaviour.

    filename  Specifies that a FileHandler be created, using the specified
              filename, rather than a StreamHandler.
    filemode  Specifies the mode to open the file, if filename is specified
              (if filemode is unspecified, it defaults to 'a').
    format    Use the specified format string for the handler.
    datefmt   Use the specified date/time format.
    style     If a format string is specified, use th
              type of format string (possible values
              %-formatting, :meth:`str.format` and :c
              - defaults to '%').
    level     Set the root logger level to the specif
    stream    Use the specified stream to initialize
              that this argument is incompatible with
              are present, 'stream' is ignored.
    handlers  If specified, this should be an iterabl
              handlers, which will be added to the ro
              in the list which does not have a forma
              assigned the formatter created in this
    force     If this keyword  is specified as true,
              attached to the root logger are removed and closed, before
              carrying out the configuration as specified by the other
              arguments.
    encoding  If specified together with a filename, this encoding is passed to
              the created FileHandler, causing it to be used when the file is
              opened.
    errors    If specified together with a filename, this value is passed to the
              created FileHandler, causing it to be used when the file is
              opened in text mode. If not specified, the default value is
              `backslashreplace`.
```

Logging Levels

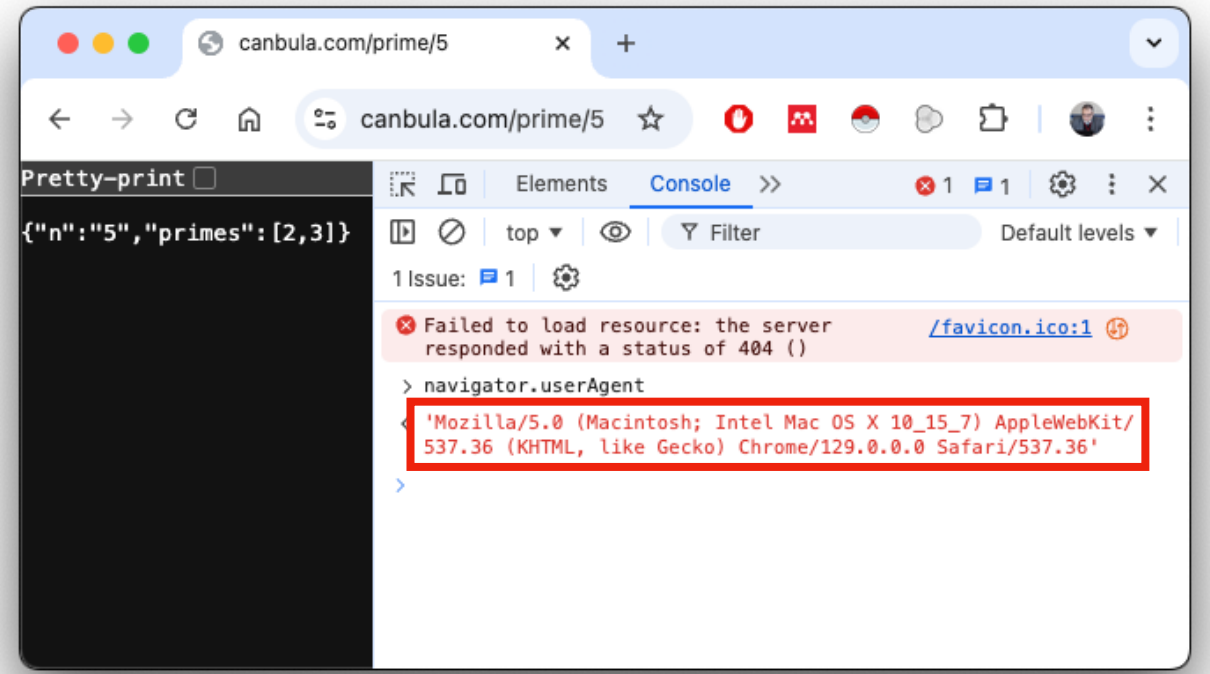
The numeric values of logging levels are given in the following table. These are primarily of interest if you want to define your own levels, and need them to have specific values relative to the predefined levels. If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

Level	Numeric value	What it means / When to use it
`logging.NOTSET`	0	When set on a logger, indicates that ancestor loggers are to be consulted to determine the effective level. If that still resolves to `NOTSET`, then all events are logged. When set on a handler, all events are handled.
`logging.DEBUG`	10	Detailed information, typically only of interest to a developer trying to diagnose a problem.
`logging.INFO`	20	Confirmation that things are working as expected.
`logging.WARNING`	30	An indication that something unexpected happened, or that a problem might occur in the near future (e.g. 'disk space low'). The software is still working as expected.
`logging.ERROR`	40	Due to a more serious problem, the software has not been able to perform some function.
`logging.CRITICAL`	50	A serious error, indicating that the program itself may be unable to continue running.

```
logging.basicConfig(
    format="%(levelname)s @ %(asctime)s : %(message)s",
    datefmt="%d.%m.%Y %H:%M:%S",
    level=logging.INFO,
    handlers=[
        logging.FileHandler("requests.log", mode="w"),
        logging.StreamHandler()
    ],
)
```

Wrap-up: Use what we have learned so far in a project!

Solving the problems of requests module




```
class Session(SessionRedirectMixin)
 |  A Requests session.
 |
 |  Provides cookie persistence, connection-
 |
 |  Basic Usage::
 |
 |    >>> import requests
 |    >>> s = requests.Session()
 |    >>> s.get('https://httpbin.org/get')
 |    <Response [200]>
 |
 |  Or as a context manager::
 |
 |    >>> with requests.Session() as s:
 |    ...     s.get('https://httpbin.org/get
 |    <Response [200]>
 |
 |  Method resolution order:
 |      Session
 |      SessionRedirectMixin
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __enter__(self)
 |
 |  __exit__(self, *args)
 |
 |  __getstate__(self)
 |      Helper for pickle.
 |
 |  __init__(self)
 |      Initialize self.  See help(type(self
 |
 |  __setstate__(self, state)
```

```
 |  request(self, method, url, params=None, data=None, headers=None, cookies=None,
=None, timeout=None, allow_redirects=True, proxies=None, hooks=None, stream=None,
=None, json=None)
 |      Constructs a :class:`Request <Request>`, prepares it and sends it.
 |      Returns :class:`Response <Response>` object.
 |
 |      :param method: method for the new :class:`Request` object.
 |      :param url: URL for the new :class:`Request` object.
 |      :param params: (optional) Dictionary or bytes to be sent in the query
 |          string for the :class:`Request`.
 |      :param data: (optional) Dictionary, list of tuples, bytes, or file-like
 |          object to send in the body of the :class:`Request`.
 |      :param json: (optional) json to send in the body of the
 |          :class:`Request`.
 |      :param headers: (optional) Dictionary of HTTP Headers to send with the
 |          :class:`Request`.
 |      :param cookies: (optional) Dict or CookieJar object to send with the
 |          :class:`Request`.
 |      :param files: (optional) Dictionary of ``'filename': file-like-objects``
 |          for multipart encoding upload.
 |      :param auth: (optional) Auth tuple or callable to enable
 |          Basic/Digest/Custom HTTP Auth.
 |      :param timeout: (optional) How long to wait for the server to send
 |          data before giving up, as a float, or a :ref:`(connect timeout,
 |          read timeout) <timeouts>` tuple.
 |      :type timeout: float or tuple
 |      :param allow_redirects: (optional) Set to True by default.
 |      :type allow_redirects: bool
 |      :param proxies: (optional) Dictionary mapping protocol or protocol and
 |          hostname to the URL of the proxy.
 |      :param stream: (optional) whether to immediately download the response
 |          content. Defaults to ``False``.
 |      :param verify: (optional) Either a boolean, in which case it controls whet
 |          the server's TLS certificate, or a string, in which case it must be a
 |          to use. Defaults to ``True``. When set to
 |          ests will accept any TLS certificate presented by
 |          will ignore hostname mismatches and/or expired
 |          certificates, which will make your application vulnerable to
 |          man-in-the-middle (MitM) attacks. Setting verify to ``False``
 |          may be useful during local development or testing.
 |      :param cert: (optional) if String, path to ssl client cert file (.pem).
 |          If Tuple, ('cert', 'key') pair.
 |      :rtype: requests.Response
```

Context Manager

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

Wrap-up: Use what we have learned so far in a project!

Time to go asynchronous!

```
Python 3.12.7 (main, Oct  1 2024, 02:05:46) [Clang 16.0.0 (clang-1600.0.26.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import aiohttp
>>> print([s for s in dir(aiohttp) if not s.startswith("_")])
['AsyncIterablePayload', 'AsyncResolver', 'BadCont
onnector', 'BasicAuth', 'BodyPartReader', 'Buffere
pProxy', 'ClientConnectionError', 'ClientConnectio
nectorDNSError', 'ClientConnectorError', 'ClientCo
lientOSError', 'ClientPayloadError', 'ClientProxyC
ResponseError', 'ClientSSLError', 'ClientSession'
eoutError', 'ContentTypeError', 'CookieJar', 'Data
', 'ETag', 'EofStream', 'Fingerprint', 'FlowContro
rnWebWorker', 'HttpVersion', 'HttpVersion10', 'Htt
ntError', 'InvalidUrlRedirectClientError', 'JsonPa
nector', 'NonHttpUrlClientError', 'NonHttpUrlRedir
ientError', 'RequestInfo', 'ServerConnectionError'
ServerTimeoutError', 'SocketTimeoutError', 'Stream
, 'TextIOPayload', 'ThreadedResolver', 'TooManyRed
TraceConnectionCreateStartParams', 'TraceConnectio
ceConnectionReuseconnParams', 'TraceDnsCacheHitPar
ams', 'TraceDnsResolveHostStartParams', 'TraceRequ
ExceptionParams', 'TraceRequestHeadersSentParams',
TraceResponseChunkReceivedParams', 'UnixConnector'
keError', 'WebSocketError', 'content_disposition_f
', 'payload_type', 'request', 'streamer']
```

```
class ClientSession(builtins.object)
 |  ClientSession(base_url: Union[str, yarl.URL, NoneType] = None, *, connector: Optional
[aiohttp.connector.BaseConnector] = None, loop: Optional[asyncio.events.AbstractEventLoop
] = None, cookies: Union[Mapping[str, Union[str, ForwardRef('BaseCookie[str]'), ForwardRe
f('Morsel[Any]')]], Iterable[Tuple[str, Union[str, ForwardRef('BaseCookie[str]'), Forward
Ref('Morsel[Any]')]]], ForwardRef('BaseCookie[str]'), NoneType] = None, headers: Union[Ma
pping[str, str], Mapping[multidict._multidict.istr, str], multidict._multidict.CIMultiDic
t, multidict._multidict.CIMultiDictProxy, Iterable[Tuple[Union[str, multidict._multidict.
istr], str]], NoneType] = None, skip_auto_headers: Optional[Iterable[str]] = None, auth:
Optional[aiohttp.helpers.BasicAuth] = None, json_serialize: Callable[[Any], str] = <funct
ion dumps at 0x103116980>, request_class: Type[aiohttp.client_reqrep.ClientRequest] = <cl
ass 'aiohttp.client_reqrep.ClientRequest'>, response_class: Type[aiohttp.client_reqrep.Cl
ientResponse] = <class 'aiohttp.client_reqrep.ClientResponse'>, ws_response_class: Type[a
iohttp.client_ws.ClientWebSocketResponse] = <class 'aiohttp.client_ws.ClientWebSocketResp
onse'>, version: aiohttp.http_writer.HttpVersion = HttpVersion(major=1, minor=1), cookie_
jar: Optional[aiohttp.abc.AbstractCookieJar] = None, connector_owner: bool = True, raise_
for_status: Union[bool, Callable[[aiohttp.client_reqrep.ClientResponse], Awaitable[NoneTy
pe]]] = False, read_timeout: Union[float, aiohttp.helpers._SENTINEL] = <_SENTINEL.sentine
l: 1>, conn_timeout: Optional[float] = None, timeout: Union[object, aiohttp.client.Client
Timeout] = <_SENTINEL.sentinel: 1>, auto_decompress: bool = True, trust_env: bool = False
, requote_redirect_url: bool = True, trace_configs: Optional[List[aiohttp.tracing.TraceCo
nfig]] = None, read_bufsize: int = 65536, max_line_size: int = 8190, max_field_size: int
= 8190, fallback_charset_resolver: Callable[[aiohttp.client_reqrep.ClientResponse, bytes]
, str] = <function ClientSession.<lambda> at 0x103cea8e0>) -> None
 |
 |  First-class interface for making HTTP requests.
 |
 |  Methods defined here:
 |
 |  async __aenter__(self) -> 'ClientSession'
 |
 |  async __aexit__(self, exc_type: Optional[Type[BaseException]], exc_val: Optional[Base
Exception], exc_tb: Optional[traceback]) -> None
 |
 |  __del__(self, _warnings: Any = <module 'warnings' from '/opt/homebrew/Cellar/python@3
.12/3.12.7_1/Frameworks/Python.framework/Versions/3.12/lib/python3.12/warnings.py'>) -> N
one
 |
 |  __enter__(self) -> None
 |
 |  __exit__(self, exc_type: Optional[Type[BaseException]], exc_val: Optional[BaseExcepti
on], exc_tb: Optional[traceback]) -> None
 |
 |  __init__(self, base_url: Union[str, yarl.URL, NoneType] = None, *, connector: Optiona
l[aiohttp.connector.BaseConnector] = None, loop: Optional[asyncio.events.AbstractEventLoo
p] = None, cookies: Union[Mapping[str, Union[str, ForwardRef('BaseCookie[str]'), ForwardR
ef('Morsel[Any]')]], Iterable[Tuple[str, Union[str, ForwardRef('BaseCookie[str]'), Forwar
dRef('Morsel[Any]')]]], ForwardRef('BaseCookie[str]'), NoneType] = None, headers: Union[M
apping[str, str], Mapping[multidict._multidict.istr, str], multidict._multidict.CIMultiDi
ct, multidict._multidict.CIMultiDictProxy, Iterable[Tuple[Union[str, multidict._multidict
.istr], str]], NoneType] = None, skip_auto_headers: Optional[Iterable[str]] = None, auth:
 Optional[aiohttp.helpers.BasicAuth] = None, json_serialize: Callable[[Any], str] = <func
tion dumps at 0x103116980>, request_class: Type[aiohttp.client_reqrep.ClientRequest] = <c
lass 'aiohttp.client_reqrep.ClientRequest'>, response_class: Type[aiohttp.client_reqrep.C
lientResponse] = <class 'aiohttp.client_reqrep.ClientResponse'>, ws_response_class: Type[
aiohttp.client_ws.ClientWebSocketResponse] = <class 'aiohttp.client_ws.ClientWebSocketRes
ponse'>, version: aiohttp.http_writer.HttpVersion = HttpVersion(major=1, minor=1), cookie
_jar: Optional[aiohttp.abc.AbstractCookieJar] = None, connector_owner: bool = True, raise
_for_status: Union[bool, Callable[[aiohttp.client_reqrep.ClientResponse], Awaitable[NoneT
ype]]] = False, read_timeout: Union[float, aiohttp.helpers._SENTINEL] = <_SENTINEL.sentin
el: 1>, conn_timeout: Optional[float] = None, timeout: Union[object, aiohttp.client.Clien
tTimeout] = <_SENTINEL.sentinel: 1>,
e, requote_redirect_url: bool = True,
onfig]] = None, read_bufsize: int = 6
 = 8190, fallback_charset_resolver: C
], str] = <function ClientSession.<la
```

```
request(method: str, url: Union[str, yarl.URL], *, p
f('Mapping[str, QueryVariable]'), ForwardRef('Sequen
ne, data: Any = None, json: Any = None, headers: Uni
ct._multidict.istr, str], multidict._multidict.CIMul
ictProxy, Iterable[Tuple[Union[str, multidict._multi
skip_auto_headers: Optional[Iterable[str]] = None, a
th] = None, allow_redirects: bool = True, max_redire
] = None, chunked: Optional[bool] = None, expect100:
onal[bool] = None, read_until_eof: bool = True, prox
one, proxy_auth: Optional[aiohttp.helpers.BasicAuth]
t.ClientTimeout, object] = <_SENTINEL.sentinel: 1>,
, ForwardRef('BaseCookie[str]'), ForwardRef('Morsel[
tr, ForwardRef('BaseCookie[str]'), ForwardRef('Morse
tr]'), NoneType] = None, version: aiohttp.http_write
minor=1), connector: Optional[aiohttp.connector.Base
onal[int] = None, loop: Optional[asyncio.events.Abst
 int = 8190, max_field_size: int = 8190) -> aiohttp.
    Constructs and sends a request.

    Returns response object.
    method - HTTP method
    url - request url
    params - (optional) Dictionary or bytes to be se
      string of the new request
    data - (optional) Dictionary, bytes, or file-lik
      send in the body of the request
    json - (optional) Any json compatible python obj
    headers - (optional) Dictionary of HTTP Headers to send with
      the request
    cookies - (optional) Dict object to send with the request
    auth - (optional) BasicAuth named tuple represent HTTP Basic Auth
    auth - aiohttp.helpers.BasicAuth
    allow_redirects - (optional) If set to False, do not follow
      redirects
    version - Request HTTP v
    compress - Set to True if request has to be compressed
       with deflate encoding.
    chunked - Set to chunk size for chunked transfer encoding.
    expect100 - Expect 100-continue response from server.
    connector - BaseConnector sub-class instance to support
       connection pooling.
    read_until_eof - Read response until eof if response
       does not have Content-Length header.
    loop - Optional event loop.
    timeout - Optional ClientTimeout settings structure, 5min
       total timeout by default.
```

Asynchronous Context Manager

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

https://fo[illegible]m/r/Nveiz7NFBC

in-class quiz

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.

Week06/timer_firstname_lastname.py

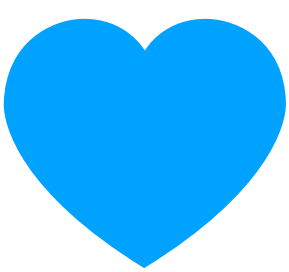
Timer

- ☑ Create a class Timer that measures the time taken by the block of code it manages.
- ☑ Timer class must be a context manager.
- ☑ The class must have two public attributes start_time and end_time, which are for the starting and the ending times, respectively.

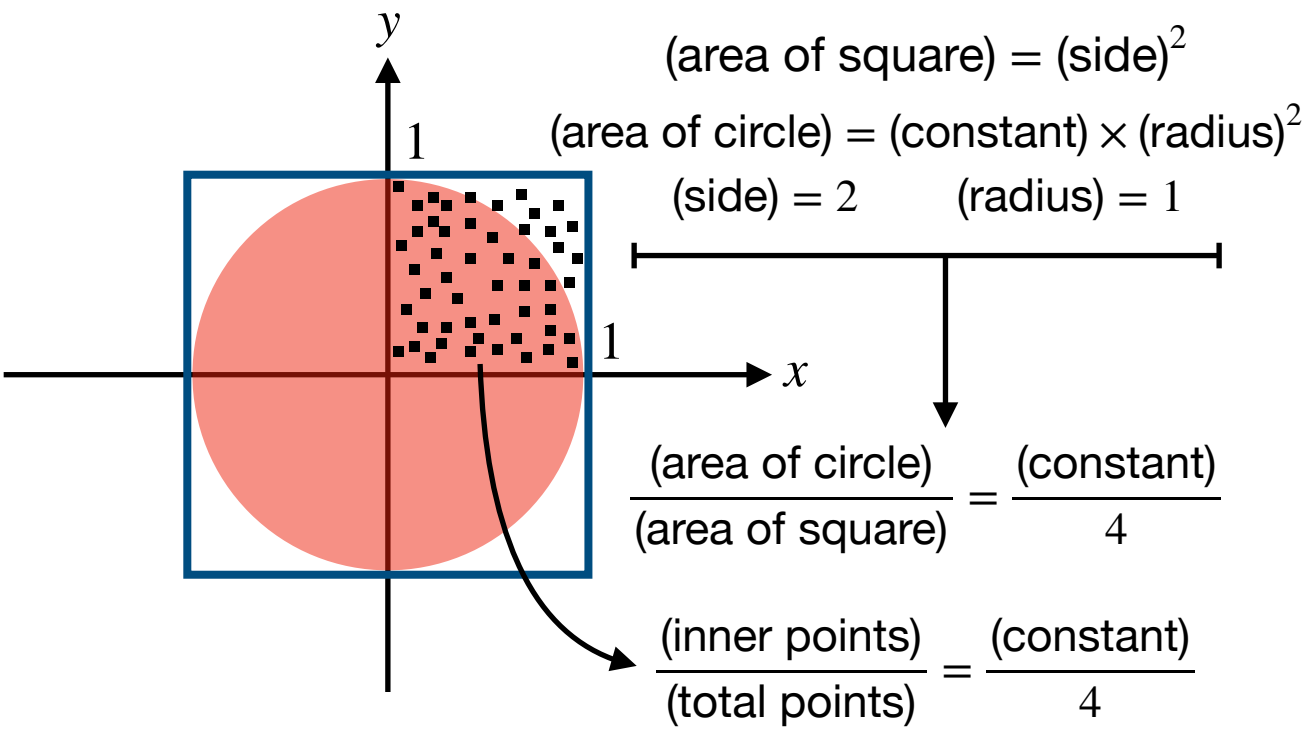
Rules for your pull requests

- ✓ 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.

Good Luck
for your midterm!



Monte Carlo Estimation of Pi



$$\text{estimated value of } \pi = 4 \times \frac{\text{(inner points)}}{\text{(total points)}}$$

```python
def next_pi():
    total = 0
    inner = 0
    while True:
        x = random.random()
        y = random.random()
        if x**2 + y**2 <= 1:
            inner += 1
        total += 1
        yield 4 * inner / total
```

increase the number of points to increase the precision

Creating Threads

From a Function

```
def thread_1():
    print("Thread 1")

t = threading.Thread(target=thread_1)
t.start()
```

From a Class

```
class Thread2(threading.Thread):
    def __init__(self):
        super().__init__()

    def run(self):
        print("Thread 2")

t = Thread2()
t.start()
```

with Arguments

```
def thread_3(s):
    print(f"Thread 3: {s}")

t = threading.Thread(target=thread_3, args=("Value",))
t.start()
```

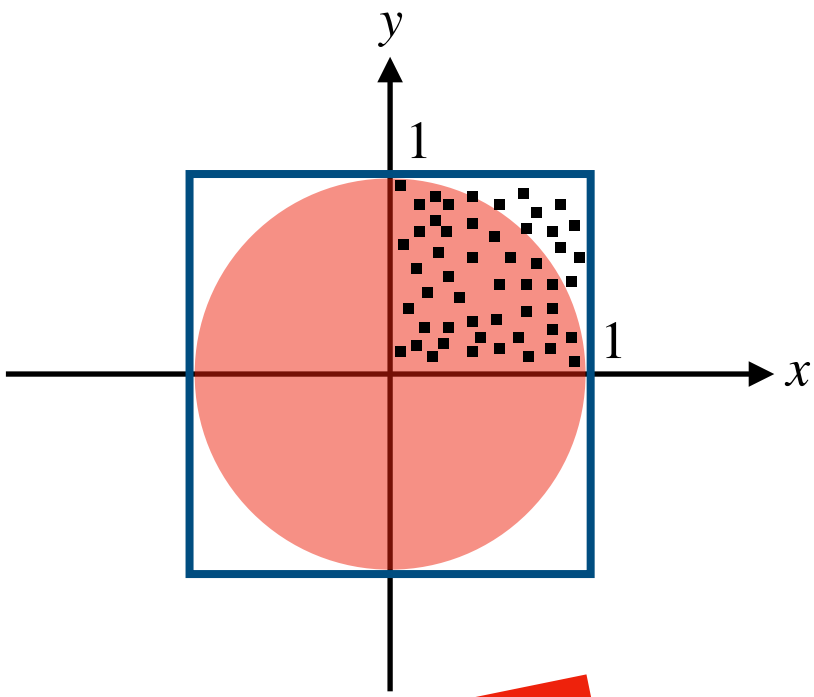
Synchronization

```
def thread_4_1():
    print("This is thread 4_1")
    time.sleep(1)
    print("Thread 4_1 is done")

def thread_4_2():
    print("This is thread 4_2")
    time.sleep(3)
    print("Thread 4_2 is done")

def thread_4_3():
    print("This is thread 4_3")
    time.sleep(1)
    print("Thread 4_3 is done")

t1 = threading.Thread(target=thr
t2 = threading.Thread(target=thr
t3 = threading.Thread(target=thr
t1.start()
t2.start()
t1.join()
t2.join()
t3.start()
```

Daemon Threads

```
def thread_5():
    print("This is a daemon thread")
    while True:
        time.sleep(1)
        print("I am still alive")
    print("I will never be printed")

def thread_6():
    print("This is a non-daemon thread")
    time.sleep(5)
    print("I am done therefore the program will exit")

t1 = threading.Thread(target=thread_5)
t1.daemon = True
# t1 = threading.Thread(target=thread_5, daemon=True)
t2 = threading.Thread(target=thread_6)
t1.start()
t2.start()
```

Revisit the pi estimation problem with threads



in-class quiz

https://fo…/r/xxi7cbFjpA

Week07/threaded_firstname_lastname.py

threaded

- Create a decorator that creates n number of threads from a function
- Decorator must accept an integer argument: n
- The decorator must create, start and then finally synchronize the threads by waiting them to finish

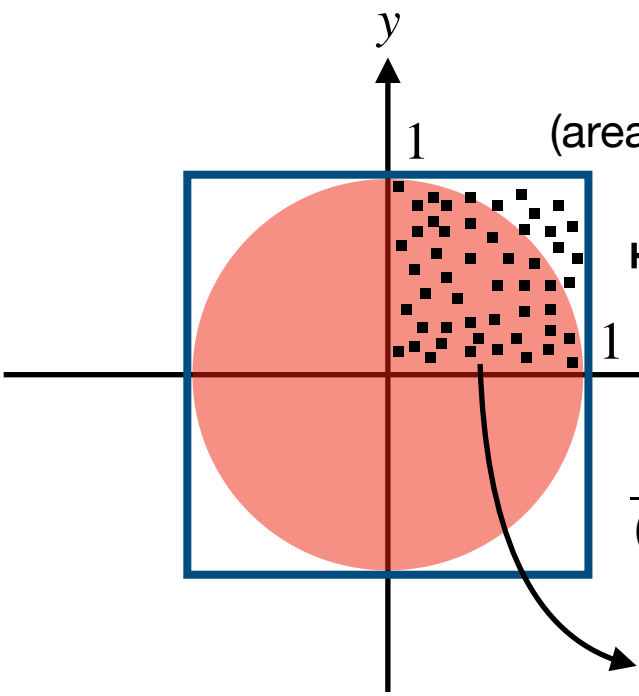
Rules for your pull requests

✓ 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.

Monte Carlo Estimation of Pi



$$(\text{area of square}) = (\text{side})^2$$

$$(\text{area of circle}) = (\text{constant}) \times (\text{radius})^2$$

$$(\text{side}) = 2 \qquad (\text{radius}) = 1$$

$$\frac{(\text{area of circle})}{(\text{area of square})} = \frac{(\text{constant})}{4}$$

$$\frac{(\text{inner points})}{(\text{total points})} = \frac{(\text{constant})}{4}$$

$$\text{estimated value of } \pi = 4 \times \frac{(\text{inner points})}{(\text{total points})}$$

Embarrassingly Parallel

A problem type, which its solution requires very little or even no effort to parallelize. The key point is that there is no need of communication between the tasks.

Atomic Part

```python
def generate_points(n: int) -> int:
    inner: int = 0
    for _ in range(n):
        x = random.random()
        y = random.random()
        if (x**2 + y**2) <= 1:
            inner += 1
    return inner
```

Atomic Part

```
def generate_points(n: int) -> int:
    inner: int = 0
    for _ in range(n):
        x = random.random()
        y = random.random()
        if (x**2 + y**2) <= 1:
            inner += 1
    return inner
```

Convert Atomic Operation to Thread

```
class PiEstimatorThread(threading.Thread):
    def __init__(
            self,
            number_of_points: int = 10000,
            name: str = None
    ) -> None:
        super().__init__()
        self.number_of_points: int = number_of_points
        self.name: str = name
        self.inner: int = 0

    def generate_points(self):
        for _ in range(self.number_of_points):
            x = random.random()
            y = random.random()
            if (x**2 + y**2) <= 1:
                self.inner += 1

    def run(self) -> None:
        self.generate_points()
```

Create another thread to generate many threads from atomic operation class and coordinate to complete the solution

Creator Thread for Atomic Operations

```python
class PiEstimator(threading.Thread):
    def __init__(
        self,
        accuracy: float = 1.0e-5,
        number_of_threads: int = 1,
        chunk_size: int = 10000,
        name: str = "PiEstimator",
    ) -> None:
        super().__init__()
        self.desired_accuracy: float = accuracy
        self.number_of_threads: int = number_of_threads
        self.chunk_size: int = chunk_size
        self.name: str = name
        self.total: int = 0
        self.inner: int = 0
        self.threads = []
        self.generated_threads: int = 0

    def pi(self) -> float:
        try:
            return float((self.inner / self.total) * 4)
        except ZeroDivisionError:
            return 0.0

    def accuracy(self) -> float:
        return abs(self.pi() - np.pi)

    def run(self):
        while self.accuracy() > self.desired_accuracy:
            for _ in range(self.number_of_threads):
                thread = PiEstimatorThread(
                    self.chunk_size,
                    name=f"Generator-{self.generated_threads}",
                )
                self.generated_threads += 1
                thread.start()
                self.threads.append(thread)
            for thread in self.threads:
                thread.join()
                self.inner += thread.inner
                self.total += thread.number_of_points
            self.threads = []

    def join(self):
        super().join()
        print(f"Final Estimate of Pi: {self.pi()}")
        print(f"Accuracy: {self.accuracy()}")
        print(f"Total Number of Points Inside the Circle: {self.inner}")
        print(f"Total Number of Points Generated: {self.total}")
        print(f"Total Number of Generated Threads: {self.generated_threads}")
```

Condition to finalize the solution

The maximum number of threads to run simultaneously

The minimum unit job for each atomic operation

Create a Main Thread

```python
def main() -> None:
    desired_accuracy = 1.0e-10
    number_of_threads = 8
    chunk_size = 1000000
    pi_estimator = PiEstimator(
        accuracy=desired_accuracy,
        number_of_threads=number_of_threads,
        chunk_size=chunk_size,
    )
    pi_estimator.start()
    pi_estimator.join()
```

```
  0[||||||||                 17.3%]   4[||||||                 11.5%]
  1[|||||||                  14.0%]   5[||||||                 12.1%]
  2[||||||                   10.7%]   6[||||||                 12.7%]
  3[||||||                   11.4%]   7[|||||||                15.4%]
Mem[||||||||||||||||||||| 1.40G/7.66G] Tasks: 6, 16 thr, 0 kthr; 3 running
Swp[                       0K/1024M] Load average: 0.68 0.47 0.23
                                     Uptime: 00:08:12

  Main   I/O
  PID USER      PRI  NI  VIRT   RES   SHR S  CPU%▽MEM%   TIME+  Command
  422 root       20   0  986M 42244 14336 S  101.3  0.5  0:23.14 python estimate_pi_w_gil.py
  433 root       20   0  986M 42244 14336 S   14.0  0.5  0:03.06 python estimate_pi_w_gil.py
  432 root       20   0  986M 42244 14336 S   12.0  0.5  0:03.06 python estimate_pi_w_gil.py
  436 root       20   0  986M 42244 14336 S   12.7  0.5  0:02.62 python estimate_pi_w_gil.py
  435 root       20   0  986M 42244 14336 S    7.3  0.5  0:02.83 python estimate_pi_w_gil.py
  431 root       20   0  986M 42244 14336 S   16.0  0.5  0:02.70 python estimate_pi_w_gil.py
  437 root       20   0  986M 42244 14336 S   12.7  0.5  0:02.64 python estimate_pi_w_gil.py
  438 root       20   0  986M 42244 14336 S   16.7  0.5  0:02.98 python estimate_pi_w_gil.py
  434 root       20   0  986M 42244 14336 R   10.0  0.5  0:02.46 python estimate_pi_w_gil.py
```

Only one thread is running a time

GIL: Global Interpreter Lock

It is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecode at once. This lock can be a significant limitation for CPU-bound problems but it is necessary mainly because CPython's memory management is not thread-safe.

Numba: Just-In-Time (JIT) Compiler for Python

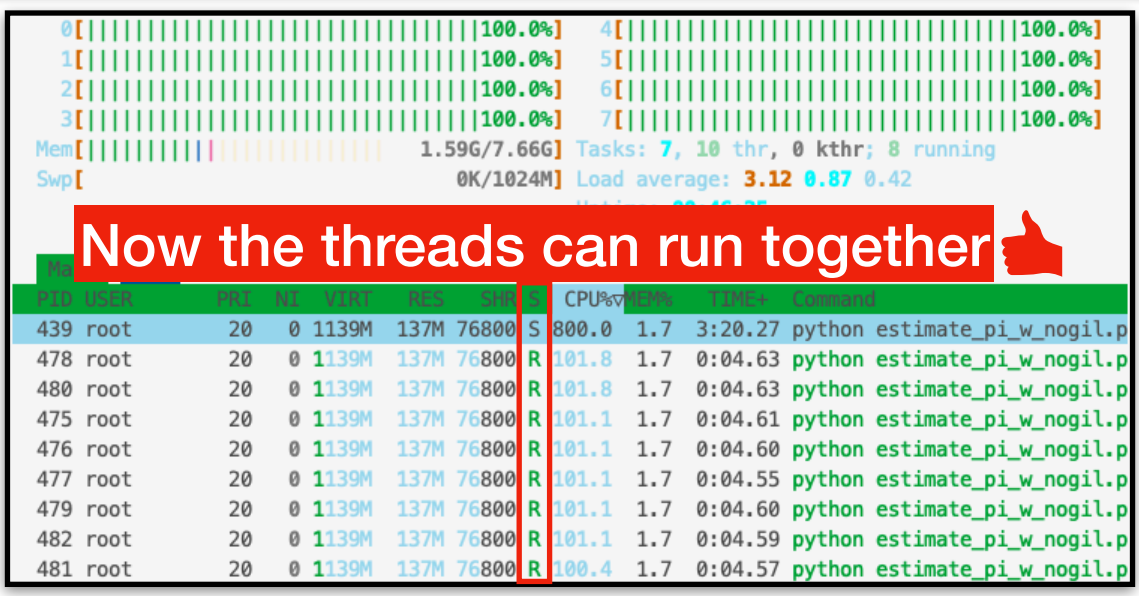
```
from numba import jit
```

→ **Numba: Just-In-Time Compiler**

```
class PiEstimatorThread(threading.Thread):
    def __init__(
            self,
            number_of_points: int = 10000,
            name: str = None
    ) -> None:
        super().__init__()
        self.number_of_points: int = number_of_points
        self.name: str = name
        self.inner: int = 0

    @staticmethod
    @jit(nopython=True, nogil=True)
    def generate_points(n):
        inner = 0
        for _ in range(n):
            x = random.random()
            y = random.random()
            if (x**2 + y**2) <= 1:
                inner += 1
        return inner

    def run(self) -> None:
        self.inner = self.generate_points(self.number_of_points)
```

The method must be static to operate independently of instance-specific data. It should be like a standalone function.

With these options, Numba compiles the function, therefore, it runs entirely without the Python interpreter.



Simple Counter

```
class Counter:
    def __init__(self):
        self.count = 0

    def increase(self):
        self.count += 1
```

Test with Threads

```
def main(n: int, m: int):
    counter = Counter()
    threads = []
    for _ in range(m):
        t = IncreaseCounter(counter, n)
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
    print(f"Expected: {n*m},
          Actual: {counter.count}")
    return counter.count
```

Increase the Counter

```
class IncreaseCounter(threading.Thread):
    def __init__(self, counter, n):
        super().__init__()
        self.counter = counter
        self.n = n

    def run(self):
        for _ in range(self.n):
            self.counter.increase()
```

```
Expected: 4, Actual: 4
Success for size 1
Expected: 40, Actual: 40
Success for size 10
Expected: 400, Actual: 400
Success for size 100
Expected: 4000, Actual: 4000
Success for size 1000
Expected: 40000, Actual: 40000
Success for size 10000
Expected: 400000, Actual: 271208
Failed for size 100000
Expected: 4000000, Actual: 1685287
Failed for size 1000000
```

Unpredictable Results

What's under the hood when we change the value of a variable

```
counter = 0
counter += 1
```

```
 0 LOAD_CONST
 2 STORE_NAME

 4 LOAD_NAME
 6 LOAD_CONST
 8 INPLACE_ADD
10 STORE_NAME
12 LOAD_CONST
14 RETURN_VALUE
```

Compiler Explorer
godbolt.org

Increasing a value sequentially

count = 0	count = 1	count = 2	count = 3
LOAD_NAME	LOAD_NAME	LOAD_NAME	LOAD_NAME
LOAD_CONST	LOAD_CONST	LOAD_CONST	LOAD_CONST
INPLACE_ADD	INPLACE_ADD	INPLACE_ADD	INPLACE_ADD
STORE_NAME	STORE_NAME	STORE_NAME	STORE_NAME
LOAD_CONST	LOAD_CONST	LOAD_CONST	LOAD_CONST
RETURN_VALUE	RETURN_VALUE	RETURN_VALUE	RETURN_VALUE
count = 1	count = 2	count = 3	count = 4

Time →

Never overlaps with each other

With Multiple Threads Without a Mutex

Time

```
count = 0
LOAD_NAME
LOAD_CONST
INPLACE_ADD
STORE_NAME
LOAD_CONST
RETURN_VALUE
count = 1
```

Race Condition

```
count = 1
LOAD_NAME
LOAD_CONST
INPLACE_ADD
STORE_NAME
LOAD_CONST
RETURN_VALUE
count = 2
```

```
count = 1
LOAD_NAME
LOAD_CONST
INPLACE_ADD
STORE_NAME
LOAD_CONST
RETURN_VALUE
count = 2
```

```
count = 2
LOAD_NAME
LOAD_CONST
INPLACE_ADD
STORE_NAME
LOAD_CONST
RETURN_VALUE
count = 3
```

```
count = 2
LOAD_NAME
LOAD_CONST
INPLACE_ADD
STORE_NAME
LOAD_CONST
RETURN_VALUE
count = 3
```

```
count = 3
LOAD_NAME
LOAD_CONST
INPLACE_ADD
STORE_NAME
LOAD_CONST
RETURN_VALUE
count = 4
```

```
count = 3
LOAD_NAME
LOAD_CONST
INPLACE_ADD
STORE_NAME
LOAD_CONST
RETURN_VALUE
count = 4
```

```
count = 3
LOAD_NAME
LOAD_CONST
INPLACE_ADD
STORE_NAME
LOAD_CONST
RETURN_VALUE
count = 4
```

```
count = 4
LOAD_NAME
LOAD_CONST
INPLACE_ADD
STORE_NAME
LOAD_CONST
RETURN_VALUE
count = 5
```

9 increments but count is 5

Time

Lock

```
count = 0
LOAD_NAME
LOAD_CONST
INPLACE_ADD
STORE_NAME
LOAD_CONST
RETURN_VALUE
count = 1
```

Lock

```
count = 1
LOAD_NAME
LOAD_CONST
INPLACE_ADD
STORE_NAME
LOAD_CONST
RETURN_VALUE
count = 2
```

Lock

```
count = 2
LOAD_NAME
LOAD_CONST
INPLACE_ADD
STORE_NAME
LOAD_CONST
RETURN_VALUE
count = 3
```

Using Locks

```python
class LockedCounter:
    def __init__(self):
        self.count = 0
        self.lock = threading.Lock()

    def increase(self):
        with self.lock:
            self.count += 1
```

```
Expected: 4, Actual: 4
Success for size 1
Expected: 40, Actual: 40
Success for size 10
Expected: 400, Actual: 400
Success for size 100
Expected: 4000, Actual: 4000
Success for size 1000
Expected: 40000, Actual: 40000
Success for size 10000
Expected: 400000, Actual: 400000
Success for size 100000
Expected: 4000000, Actual: 4000000
Success for size 1000000
```

Now it is safe but very slow!

https://fo om/ r/ aWZ

in-class quiz

Synchronization Concepts

Synchronization ensures that threads coordinate their actions effectively when accessing shared resources. Without proper synchronization, issues like race conditions, deadlock, and inconsistent data states can arise, leading to unpredictable program behavior.

Deadlock

When two or more threads are waiting on each other to release locks, causing an infinite waiting state.

Semaphore

A semaphore is used to limit the number of threads accessing a shared resource. A counter that decreases when a thread acquires it and increases when a thread releases it.

Barrier

It ensures that multiple threads reach a certain point in execution before any of them proceed.

Condition

It is used for synchronization by communicating between threads

Deadlock

```python
import threading

lock_a = threading.Lock()
lock_b = threading.Lock()

def thread_func1():
    with lock_a:
        print("Thread 1: Holding lock_a...")
        with lock_b:
            print("Thread 1: Acquired lock_b!")

def thread_func2():
    with lock_b:
        print("Thread 2: Holding lock_b...")
        with lock_a:
            print("Thread 2: Acquired lock_a!")

t1 = threading.Thread(target=thread_func1)
t2 = threading.Thread(target=thread_func2)

t1.start()
t2.start()

t1.join()
t2.join()
```

Synchronization Concepts

Synchronization ensures that threads coordinate their actions effectively when accessing shared resources. Without proper synchronization, issues like race conditions, deadlock, and inconsistent data states can arise, leading to unpredictable program behavior.

Deadlock

When two or more threads are waiting on each other to release locks, causing an infinite waiting state.

Semaphore

A semaphore is used to limit the number of threads accessing a shared resource. A counter that decreases when a thread acquires it and increases when a thread releases it.

Barrier

It ensures that multiple threads reach a certain point in execution before any of them proceed.

Condition

It is used for synchronization by communicating between threads

Semaphore

```python
import threading
import time

semaphore = threading.Semaphore(2)

def task(thread_id):
    print(f"T-{thread_id}: Waiting for semaphore")
    semaphore.acquire()
    print(f"T-{thread_id}: Acquired semaphore!")
    time.sleep(1)
    semaphore.release()
    print(f"T-{thread_id}: Released semaphore!")

threads = [
    threading.Thread(target=task, args=(i,))
    for i in range(4)
]

for t in threads:
    t.start()

for t in threads:
    t.join()
```

Synchronization Concepts

Synchronization ensures that threads coordinate their actions effectively when accessing shared resources. Without proper synchronization, issues like race conditions, deadlock, and inconsistent data states can arise, leading to unpredictable program behavior.

Deadlock

When two or more threads are waiting on each other to release locks, causing an infinite waiting state.

Semaphore

A semaphore is used to limit the number of threads accessing a shared resource. A counter that decreases when a thread acquires it and increases when a thread releases it.

Barrier

It ensures that multiple threads reach a certain point in execution before any of them proceed.

Condition

It is used for synchronization by communicating between threads

Barrier

```python
import threading

barrier = threading.Barrier(3)

def worker(thread_id):
    print(f"T-{thread_id}: Reached the barrier.")
    barrier.wait()
    print(f"T-{thread_id}: Passed the barrier!")

threads = [
    threading.Thread(target=worker, args=(i,))
    for i in range(3)
]

for t in threads:
    t.start()

for t in threads:
    t.join()
```

Synchronization Concepts

Synchronization ensures that threads coordinate their actions effectively when accessing shared resources. Without proper synchronization, issues like race conditions, deadlock, and inconsistent data states can arise, leading to unpredictable program behavior.

Deadlock

When two or more threads are waiting on each other to release locks, causing an infinite waiting state.

Semaphore

A semaphore is used to limit the number of threads accessing a shared resource. A counter that decreases when a thread acquires it and increases when a thread releases it.

Barrier

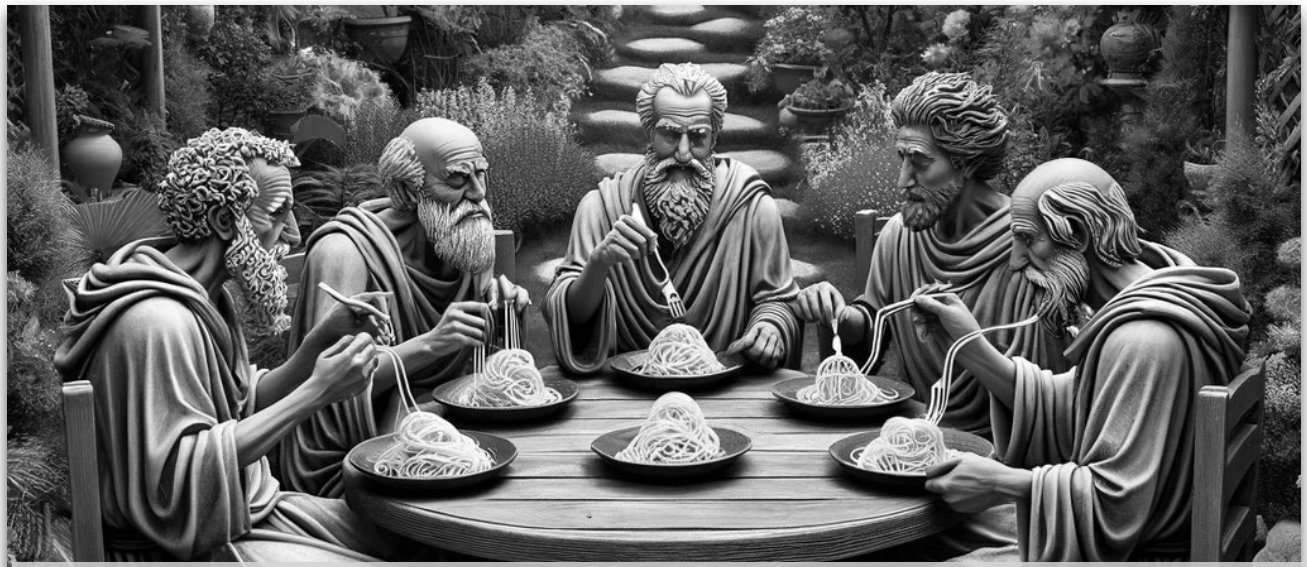
It ensures that multiple threads reach a certain point in execution before any of them proceed.

Condition

It is used for synchronization by communicating between threads

Condition

```python
import threading

condition = threading.Condition()
shared_data = []

def producer():
    with condition:
        print("Producer: Adding item to shared_data.")
        shared_data.append("data")
        condition.notify()

def consumer():
    with condition:
        print("Consumer: Waiting for data...")
        condition.wait()
        print(f"Consumer: Got {shared_data.pop()}!")

consumer_thread = threading.Thread(target=consumer)
producer_thread = threading.Thread(target=producer)

consumer_thread.start()
producer_thread.start()

consumer_thread.join()
producer_thread.join()
```



- Five philosophers, sitting around a circular table.
- There are five chairs and five plates full of spaghetti.
- Between each pair of plates, there is a single fork.
- A philosopher can only think or eat.
- To eat, a philosopher must have two forks: one from their left and one from their right.
- After eating, they put down both forks, and then they start thinking again.

The problem is to design a protocol that allows them eating

Why is it challenging? What can happen?

Implementing the Problem

The modules:
- threading, random, time

The classes:
- Philosopher (Thread)
- Fork (Custom Lock)

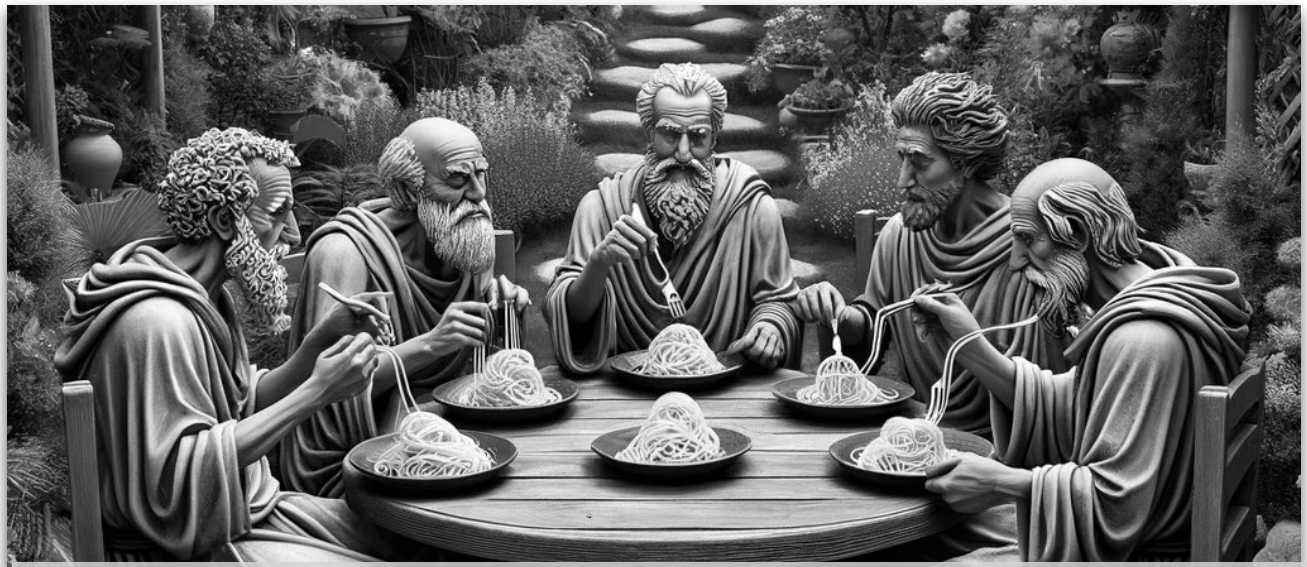
The inputs:
- n (number of philosophers)
- spaghetti (amount in integers)

Deadlock

All philosophers hold one fork and wait indefinitely for the other one.

Starvation

The solution of deadlock results in some philosophers never eat.



- Five philosophers, sitting around a circular table.
- There are five chairs and five plates full of spaghetti.
- Between each pair of plates, there is a single fork.
- A philosopher can only think or eat.
- To eat, a philosopher must have two forks: one from their left and one from their right.
- After eating, they put down both forks, and then they start thinking again.

The problem is to design a protocol that allows them eating

Why is it challenging? What can happen?

Implementing the Problem

The modules:
- threading, random, time

The classes:
- Philosopher (Thread)
- Fork (Custom Lock)

The inputs:
- n (number of philosophers)
- spaghetti (amount in integers)

Odd-Even Strategy

Philosophers are numbered from 1 to n. Philosophers with odd numbers pick up their left fork first and then their right fork. Philosophers with even numbers do the opposite.

https://f[illegible]com/[illegible]

in-class quiz

Project Assignment for 50% of your Final Grade

Cracking the Password Generator API with Asynchronous - Multi Thread - Multi Process Programming

Presentation Date 30/12/2024

```python
from flask import Flask, request, jsonify
import hashlib
import random
import string
import json


app = Flask(__name__)


def generate_password():
    password = "".join(
        random.choices(string.ascii_letters + string.digits, k=random.randint(8, 16))
    )
    return hashlib.md5(password.encode()).hexdigest()


@app.route("/get_password", methods=["GET"])
def get_password():
    password = generate_password()
    response = jsonify({"password": password})
    with open("password.json", "w") as f:
        json.dump({"password": password}, f)
    return response


@app.route("/check_password", methods=["POST"])
def check_password():
    data = request.get_json()
    password = data.get("password")
    password_hash = hashlib.md5(password.encode()).hexdigest()
    with open("password.json", "r") as f:
        stored_password = json.load(f).get("password")
    if password_hash == stored_password:
        return jsonify({"message": "Success"})
    else:
        return jsonify({"message": "Failed"})


if __name__ == "__main__":
    app.run()
```