



# PARALLEL PROGRAMMING

Assoc. Prof. Dr. Bora Canbula

Manisa Celal Bayar University  
Department of Computer Engineering



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

Tii 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**

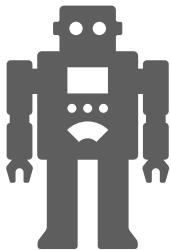
- Python 3.x (Anaconda is preferred)
- Jupyter Notebook from Anaconda
- Pycharm from JetBrains / Microsoft Visual Studio Code
- PC with a Linux distro or a Linux terminal in Windows 10/11.

**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.



### 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.



- 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.

- 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.

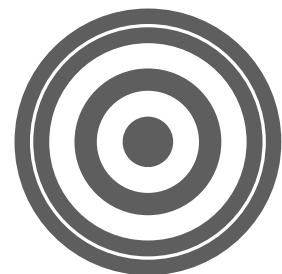
Python was created by Guido van Rossum. The name of the Python programming language comes from an old BBC television comedy sketch series called Monty Python's Flying Circus.



Guido van Rossum

## 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 translates Python code into "C" to make it run much faster than pure Python.
- **Jython** is an implementation follows only Python 2, not Python 3, written in Java.
- **PyPy** represents a Python environment written in Python-like language named RPython (Restricted Python), which is actually a subset of Python.
- **MicroPython** is an implementation of Python 3 that is optimized to run on microcontrollers.

## Start Coding with Python

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

first.py

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



<https://forms.gle/r/3qrM5Gj66X>



<https://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**

- `abs()`
- `aiter()`
- `all()`
- `anext()`
- `any()`
- `ascii()`

**E**

- `enumerate()`
- `eval()`
- `exec()`

**F**

- `filter()`
- `...`

**L**

- `len()`
- `list()`
- `locals()`

**M**

- `map()`
- `...`

**R**

- `range()`
- `repr()`
- `reversed()`
- `round()`

**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()`  
`compile()`  
`complex()`

`help()`  
**hex()**

`ord()`

`type()`

**D**

`id()`

`P`  
`pow()`  
`print()`

`V`  
`vars()`

### 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.

## Identifier Names

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

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



## keyword — Testing for Python keywords

Source code: [Lib/keyword.py](#)

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

**keyword.iskeyword(s)**

Return `True` if *s* is a Python **keyword**.

**keyword.kwlist**

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

**keyword.issoftkeyword(s)**

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

*New in version 3.9.*

**keyword.softkwlist**

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

*New in version 3.9.*



<https://forms.gle/r/VBi2yJZiX5>



<https://forms.office.com/r/RRCyEr8RE8>

## Your First Homework

ParallelProgramming Public Watch 3

master 2 branches 0 tags Go to file Add file Code

canbula tests for types and sequences 818c8da 4 minutes ago 99 commits

.github/workflows update actions 3 hours ago

Week01 add Syllabus last week

Week02 tests for types and sequences 4 minutes ago

README.md Update README for 2023 last week

ParallelProgramming / Week02 /

You need to fork this repository to propose changes.

Sorry, you're not able to edit this repository directly—you need to fork it and propose your changes from there instead.

Fork this repository

Learn more about forks

+ Create new file

Upload files

Copy path

Copy permalink

Delete directory

View options

Center content

test\_sequences.py

tests for types and :

test\_types.py

tests for types and :

ParallelProgramming / Week02

types\_bora\_canbula.py

master

Cancel changes

Commit changes...

Edit

Preview

Code 55% faster with GitHub Copilot

Spaces

2

No wrap

1 Enter file contents here

- 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

## Equality

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]

```
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
<code>+, -</code>	unary
<code>**</code>	binary
<code>*, /, //, %</code>	binary
<code>+, -</code>	binary
<code>&lt;, &lt;=, &gt;, &gt;=</code>	binary
<code>!=, ==</code>	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 occurrence)
```

### 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 occurrence 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 occurrence 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 CUMBALICA  
MANISA CELAL BAYAR UNIVERSITY

#### LISTS IN PYTHON:

```
Ordered and mutable sequence of values indexed by integers
Initializing
a_list = [] ## empty
a_list = [ ] ## empty
a_list = [3, 4, 5, 6, 7] ## filled
Finding the index of an item
a_list.index(5) ## 2 (the first occurrence)
Accessing the items
a_list[0] ## 3
a_list[1] ## 4
a_list[-1] ## 7
a_list[-2] ## 6
a_list[2:] ## [3, 6, 7]
a_list[1:2] ## [3, 4]
a_list[:4] ## [3, 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 ## [1, 4, 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 occurrence 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 = {} ## empty
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.pop(0) ## 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
Finding the index of an item
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
a_dict["name"] = "Bora", "city": "Manisa", "age": 37
Update an item
a_dict["name"] = "Bora"
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 entries in unsorted 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)
```

#### 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 occurrence)
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 entries in unsorted 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)
```



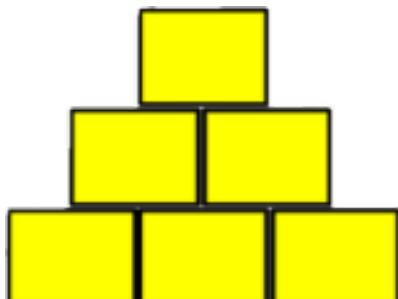
<https://forms.office.com/r/WVGNuHabIV>



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



Week03/pyramid\_first\_last.py



```
def calculate_pyramid_height(number_of_blocks):
    height = 0
    while number_of_blocks > 0:
        height += 1
        number_of_blocks -= height
    return height
```



Week03/sequences\_first\_last.py

```
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 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.")
```

### Handling All Exceptions

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

### 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)
```

### 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.")
```

### 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
```

### 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
```

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

## Parameter Kinds

PEP 362

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

## 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
<code>abs()</code> <code>aiter()</code> <code>all()</code> <code>anext()</code> <code>any()</code> <code>ascii()</code>	<code>enumerate()</code> <code>eval()</code> <code>exec()</code>	<code>len()</code> <code>list()</code> <code>locals()</code>	<code>range()</code> <code>repr()</code> <code>reversed()</code> <code>round()</code>
B	F	M	S
<code>bin()</code> <code>bool()</code> <code>breakpoint()</code> <code>bytearray()</code> <code>bytes()</code>	<code>filter()</code> <code>float()</code> <code>format()</code> <code>frozenset()</code>	<code>map()</code> <code>max()</code> <code>memoryview()</code> <code>min()</code>	<code>set()</code> <code>setattr()</code> <code>slice()</code> <code>sorted()</code> <code>staticmethod()</code> <code>str()</code> <code>sum()</code> <code>super()</code>
C	G	N	T
<code>callable()</code> <code>chr()</code> <code>classmethod()</code> <code>compile()</code> <code>complex()</code>	<code>getattr()</code> <code>globals()</code>	<code>next()</code>	<code>tuple()</code> <code>type()</code>
D	H	O	V
<code>delattr()</code> <code>dict()</code> <code>dir()</code> <code>divmod()</code>	<code>hasattr()</code> <code>hash()</code> <code>help()</code> <code>hex()</code>	<code>object()</code> <code>oct()</code> <code>open()</code> <code>ord()</code>	<code>vars()</code>
I	P	Z	
<code>id()</code> <code>input()</code> <code>int()</code> <code>isinstance()</code> <code>issubclass()</code> <code>iter()</code>	<code>pow()</code> <code>print()</code> <code>property()</code>	<code>zip()</code>	<code>__import__()</code>

## 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://in-class.office.com/r/1LvXFpEKU>



## 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})
```



## 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.