Contents

Welcome to Python Workshop. 🐍

Hello Pythoneer! Want to learn Python 🐍 with having fun? You have come to the right place ❤️.

Made with ❤️.

1. Getting Started with Python 🐍

It's pretty easy to start with Python Language 🐍. We would be using Python >= 3.9 in this Repository as of now 🙂

1. Download Python
2. Pull the docker image naveen8/pythonworkshop or build a Docker image by using the Dockerfile present in our Repository and run the container out of it which comes bundled with everything to run the code present in our repository 🚀.

Lets check the version of Python we are using. We have 2 ways to know this.

1. Open the cmd or terminal and execute **python –version**
2. Using Python's builtin sys module

```
import sys

print(sys.version)
```

```
3.9.5 (default, May 11 2021, 08:20:37)
[GCC 10.3.0]
```

2. Creating variables and assigning values

Python is a Dynamically typed language. It means based on the value we assign to a variable, it sets the datatype to it.

Now the question is "How do we assign a value to a variable?🤔". It's pretty easy.

```
<variable_name> = <value>
```

We have a big list of data types that come as builtins in Python.

- None
- bytes
- int
- bool
- float
- complex
- string
- tuple
- list
- set
- dict

Apart from the above prominent data types, we have a few other data types like namedtuple, frozensets, etc..

Let's create examples for the above data types, will be little bored in just seeing the examples. We would be covering in depth about these data types in upcoming chapters :)

Few things to know before getting into the examples:😉

1. `print` function is used to print the data on to the console. We used `f` inside the print function which is used to format the strings as `{}`, these are known as f-strings.
2. `type` function is used to find the type of the object or datatype.

```
# None
none_datatype = None
print(f"The type of none_datatype is {type(none_datatype)}")
```

```
The type of none_datatype is <class 'NoneType'>
```

```
# int
int_datatype = 13
print(f"The type of int_datatype is {type(int_datatype)}")
```

```
The type of int_datatype is <class 'int'>
```

```
# bytes
bytes_datatype = b"Hello Python!"
print(f"The type of bytes_datatype is {type(bytes_datatype)}")
```

```
The type of bytes_datatype is <class 'bytes'>
```

```
# bool
# bool datatype can only have either True or False. Integer value of True is 1 and
False is 0.
bool_datatype = True
print(f"The type of bool_datatype is {type(bool_datatype)}")
```

```
The type of bool_datatype is <class 'bool'>
```

```
# float
float_datatype = 3.14
print(f"The type of float_datatype is {type(float_datatype)}")
```

```
The type of float_datatype is <class 'float'>
```

```
# complex
complex_datatype = 13 + 5j
print(f"The type of complex_datatype is {type(complex_datatype)}")
```

```
The type of complex_datatype is <class 'complex'>
```

```
# str
str_datatype = "Hey! Welcome to Python."
print(f"The type of str_datatype is {type(str_datatype)}")
```

```
The type of str_datatype is <class 'str'>
```

```
# tuple
tuple_datatype = (None, 13, True, 3.14, "Hey! Welcome to Python.")
print(f"The type of tuple_datatype is {type(tuple_datatype)}")
```

```
The type of tuple_datatype is <class 'tuple'>
```

```
# list
list_datatype = [None, 13, True, 3.14, "Hey! Welcome to Python."]
print(f"The type of list_datatype is {type(list_datatype)}")
```

```
The type of list_datatype is <class 'list'>
```

```
# set
set_datatype = {None, 13, True, 3.14, "Hey! Welcome to Python."}
print(f"The type of set_datatype is {type(set_datatype)}")
```

```
The type of set_datatype is <class 'set'>
```

```
# dict
dict_datatype = {
    "language": "Python",
    "Inventor": "Guido Van Rossum",
    "release_year": 1991,
}
print(f"The type of dict_datatype is {type(dict_datatype)}")
```

```
The type of dict_datatype is <class 'dict'>
```

2.1. Tidbits

The thing which I Love the most about Python is the dynamic typing, Due to this we might not know what are the types of parameters we might pass to a function or method. If you pass any other type of object as a parameter, **boom** you might see Exceptions raised during the runtime👻. Let's remember that **With great power comes great responsibility** 🕷️

To help the developers with this, from Python 3.6 we have Type Hints(PEP-484).

We will get through these in the coming chapters. Stay tuned 😇

3. Python Keywords and allowed Variable names

```
# To retrieve the python keyword list, we can use the keyword built-in package.
import keyword
```

Let's print the keywords present.

keyword.kwlist returns python's keywords in a list datatype.

We are using *(starred) expression to print the values returned by keyword.kwlist each separated by "\n"(newline).

```
print(*keyword.kwlist, sep="\n")
```

```
False
None
True
__peg_parser__
and
as
assert
async
await
break
class
continue
def
del
elif
else
except
finally
for
from
global
if
import
in
is
lambda
nonlocal
not
or
pass
raise
return
try
while
with
yield
```

3.1. Variable Names

TLDR:

- Variable names shouldn't be same as that of built-in keywords.
- Variable name shouldn't start with a number or with a symbol(except "_", protected and private attributes are created using underscore, 🤔 it's better to say it as name mangling rather than protected or private. That's for a different notebook session 😀).

PS: Don't give a try naming the variable that starts with #, it would be a Python's comment, which would be neglected by the interpreter 😅.

3.1.1. Allowed Variable names

```
x = True
_x = False
x_y = "Hey Python geek!"
x9 = "alphabet_number"
# Python is a case sensitive language, so `x` is different from `X`. Let's give it a try.
X = "one more variable"
print(f"x is equal to X:{x==X}")
```

```
x is equal to X:False
```

3.1.2. Invalid Variable names

We will be using `exec` within `try-except` to catch the syntax error. 🤔 But why? Syntax errors can't be catched, well it shouldn't for good 😉. so we are using `exec` to execute the code.

`exec` takes the string argument and interprets the string as a python code.

```
# variable name starting with number.
code_string = "9x=True"
try:
    exec(code_string)
except SyntaxError as exc:
    print(f"Ouch! In the exception: {exc}")
```

```
Ouch! In the exception: invalid syntax (<string>, line 1)
```

```
# variable name starting with a symbol(other than underscore"_").
code_string = "$g = 10"
try:
    exec(code_string)
except SyntaxError as exc:
    print(f"Ouch! In the exception: {exc}")
```

```
Ouch! In the exception: invalid syntax (<string>, line 1)
```

4. Data types

`Kiddo explanation` 😇:

We might use many materials like sand, bricks, concrete to construct a house. These are basic and essential needs to have the construction done and each of them have a specific role or usage.

Likewise, we need various data types like string, boolean, integer, dictionary etc.. for the development of a code. We need to know where to use a specific data type and it's functionality.😊

We have various built-in data types that come out of the box 😎.

Data type	Mutable?
None	❌
bytes	❌
bool	❌
int	❌
float	❌
complex	❌
str	❌
tuple	❌
list	✅
set	✅
dictionary	✅

The First question we would be interested in is “What is Mutable?🤔”. If a object can be altered after its creation, then it is Mutable, else Immutable.

4.1. **None**

None is a singleton object, which represents empty or null.

4.1.1. *Example of None usage*:

In this example, Let’s try getting the environment variables 😉

We would be using the `os` module’s `getenv` method to fetch the environment variable’s value, if there isn’t that environment variable, it would be returning `None`

```
import os

# let's set a env variable first
new_environment_variable_name: str = "Python"
new_environment_variable_value: str = "1991"
os.environ[new_environment_variable_name] = new_environment_variable_value

# Now let's try to fetch a envrionment's variable value
value = os.getenv(new_environment_variable_name)
if value is None:
    print(f"There is no environment variable named {new_environment_variable_name}")
else:
    print(
        f"The value assigned for the environment variable named
{new_environment_variable_name} is {value}"
    )
```

```
The value assigned for the environment variable named Python is 1991
```

4.2. **bytes**

byte objects are the sequences of bytes, these are machine readable form and can be stored on the disk. Based on the encoding format, the bytes yield results.

bytes can be converted to string by decoding it, vice-versa is known as encoding.

bytes objects can be created by prefixing `b` before the string.

```
bytes_obj: bytes = b"Hello Python Enthusiast!"
print(bytes_obj)
```

```
b'Hello Python Enthusiast!'
```

We see that they are visually the same as string when printed. But actually they are ASCII values, for the convenience of the developer, we see them as human readable strings.

But how to see the actual representation of bytes object? 🤔 It's pretty simple 😉! We can typecast the bytes object to a list and we see each character as it's respective ASCII value.

```
print(list(bytes_obj))
```

```
[72, 101, 108, 108, 111, 32, 80, 121, 116, 104, 111, 110, 32, 69, 110, 116, 104, 117,
115, 105, 97, 115, 116, 33]
```

4.3. **bool**

bool objects have only two values: `True`✅ and `False`❌, integer equivalent of True is 1 and for False is 0

```
do_we_love_python = True
if do_we_love_python:
    print("🐍 Python too loves and takes care of you ❤")
else:
    print("🐍 Python still loves you ❤")
```

```
🐍 Python too loves and takes care of you ❤
```

PS: Boolean values in simple terms mean **Yes** for `True` and **No** for `False`

4.4. **int**

int objects are any mathematical Integers. pretty easy right 😎

```
# Integer values can be used for any integer arithmetics.
# A few simple operations are addition, subtraction, multiplication, division etc..
operand_1 = 5
operand_2 = 3
print(operand_1 + operand_2)
```

```
8
```

4.5. **float**

float objects are any rational numbers.

```
# Like integer objects float objects are used for decimal arithmetics
# A few simple operations are addition, subtraction, multiplication, division etc..
# We are typcasting integer or float value to float values explicitly.
operand_1 = 3.18
operand_2 = 6.24
print(operand_1 + operand_2)
```

```
9.42
```

4.6. **complex**

complex objects aren't so complex to understand 😉

complex objects hold a Real number and an imaginary number. While creating the complex object, we would be having a `j` beside the imaginary number.

```
operand_1 = 10 + 5j
operand_2 = 3 + 4j
print(operand_1 * operand_2)
```

```
(10+55j)
```

explanation for the above math: 😉

```
(3+4j)*(10+5j)
3(10+5j) + 4j(10+5j)
30 + 15j + 40j + 20(j*j)
30 + 15j + 40j + 20(-1)
30 + 15j + 40j - 20
30 - 20 + 15j + 40j
10 + 55j
```

4.7. **str**

string objects hold an sequence of characters.

```
my_string = "🐍 Python is cool"
print(my_string)
```

```
🐍 Python is cool
```

4.8. **tuple**

tuple object is an immutable datatype which can have any datatype objects inside it and is created by enclosing paranthesis `()` and objects are separated by a comma.

Once the tuple object is created, the tuple can't be modified, although if the objects in the tuple are mutable, they can be changed 😊

The objects in the tuple are ordered, So the objects in the tuple can be accessed by using its index ranging from 0 to (number of elements - 1).

```
# tuples are best suited for having data which doesn't change in it's lifetime.

apple_and_its_colour = ("apple", "red")
watermelon_and_its_colour = ("watermelon", "green")

language_initial_release_year = ("Golang", 2012)
language_initial_release_year = ("Angular", 2010)
language_initial_release_year = ("Python", 1990)

# We can't add new data types objects, delete the existing datatype objects, or change
the values
# of the existing objects.

# We can get the values by index.
print(
    f"{language_initial_release_year[0]} is released in
{language_initial_release_year[1]}"
)
```

```
Python is released in 1990
```

4.9. **list**

list objects are similar to tuple, the differences are the list object is mutable, so we can add or remove objects in the list even after its creation. It is created by using `[]`.

```
about_python = [
    "interpreted",
    "object-oriented",
    "dynamically typed",
    "open source",
    "high level language",
    "🐍",
    1990,
]
print(about_python)
# We can add more values to the above list. append method of list object is used to add
a new object.
# let's give a try 🤞

about_python.append("Guido Van Rossum")
print(about_python)
```

```
['interpreted', 'object-oriented', 'dynamically typed', 'open source', 'high level
language', '🐍', 1990]
['interpreted', 'object-oriented', 'dynamically typed', 'open source', 'high level
language', '🐍', 1990, 'Guido Van Rossum']
```

4.10. **set**

set objects are unordered, unindexed, non repetitive collection of objects. Mathematical set theory operations can be applied using set datatype objects. 😊 it is created by using `{}`.

PS: `{}` denotes a dictionary, we need to use `set()` for creating an empty set, there won't be this issue when creating set objects containing objects, for example: `{1,"a"}`

set objects are good for having the mathematical set operations.

```
set_obj = {6, 4, 4, 3, 10, "Python", "Python", "Golang"}
# We see that we have created a set with 8 objects.
print(set_obj)
# But when printed, we see that only 6 are present because set doesn't allow same
objects repeated.
```

```
{'Python', 3, 4, 'Golang', 6, 10}
```

4.11. **dict**

dictionary objects are used for creating key-value pairs, Here keys would be unique while values can be repeated.

The object assigned to a key can be fetched by using `<dict_obj>[key]` which raises a KeyError when no given key is found. The other way to fetch is by using `<dict_obj>.get(key)` which returns `None` by default if no key is found.

```
dict_datatype = {
    "language": "Python",
    "Inventor": "Guido Van Rossum",
    "release_year": 1991,
}
print(f"The programming language is: {dict_datatype['language']}")
# We could use get method to prevent KeyError if the given Key is not found.
result = dict_datatype.get("LatestRelease")
# Value of the result would be None as the key LatestRelease is not present in
dict_datatype
print(f"The result is: {result}")
```

```
The programming language is: Python
The result is: None
```

5. Collection Types

We have many collection types in Python, `str`, `int` objects hold only value, but coming to collection types, we can have various objects stored in the collections.

The Collection Types we have in Python are:

- Tuple
- List
- Set
- Dictionary

5.1. Tuple

A Tuple is a ordered collection of objects and it is of fixed length and immutable, so the values in the tuple can not be changed nor added or removed.

Tuples are generally used for small collections which we are sure about them from right before such as IP addresses and port numbers. Tuples are represented with paranthesis `()`

Example:

```
ip_address_port = ("127.0.0.1", 8080)
```

A tuple with a single member needs to have a trailing comma, else the type of the variable would be the datatype of the member itself.

```
# Proper way to create a single member tuple.
single_member_tuple = ("one",)
print(type(single_member_tuple))
single_member_tuple = ("one",)
print(type(single_member_tuple))
```

```
<class 'tuple'>
<class 'tuple'>
```

```
# Improper way trying to create a single member tuple.
single_member_tuple = "one"
print(type(single_member_tuple))
```

```
<class 'str'>
```

5.2. List

List collection types are similar to tuples, the only difference would be that new objects can be created, removed or object's data can be modified 😉.

```
int_list = [1, 2, 3]
string_list = ["abc", "defghi"]
```

```
# A list can be empty:
empty_list = []
```

objects in the list are not restricted to be of a particular datatype. let's see an example 👇.

```
mixed_list = [1, "abc", True, 3.14, None]
```

list can contain lists as objects too. These are called nested lists.

```
nested_list = [[1, 2, 3], ["a", "b", "c"]]
```

The objects present in the list can be accessed by the index it is placed. The index starts from 0 👻.

```
my_list = ["Iron man", "Thor", "Wonder Woman", "Wolverine", "Naruto"]
```

```
print(my_list[0])
print(my_list[1])
```

```
Iron man
Thor
```

In the `my_list`, we have 5 strings in the list, but in the below example, let's give a try to get the 100th index element which is not present in the `my_list` 🙄.

As there is no 100th element, we would be seeing an `IndexError` exception.

```
try:
    print(my_list[100])
except IndexError as exc:
    print(f"👻 Ouch! we got into IndexError exception: {exc}")
```

```
👻 Ouch! we got into IndexError exception: list index out of range
```

The question I have is, how do I get the 2nd element from the last 🤔? Should I find the length of the list and access the <length - 2>? Yup, it works 😉.

But we have one good way to do it by negative index, example: `-2`

```
# Access the 2nd element from the last.
print(my_list[-2])
```

```
Wolverine
```

5.2.1. We have a few methods of list that we can give it a try now 😎

`append`

```
# Append a new item to the list.
# We use append method of the list.
my_list.append("Zoro")
print(my_list)
```

```
['Iron man', 'Thor', 'Wonder Woman', 'Wolverine', 'Naruto', 'Zoro']
```

`remove`

```
# Remove the item present in the list.
# We use remove method of the list.
# If there's no object that we are trying to remove in the list, then ValueError would
be raised.
try:
    my_list.remove("Zoro")
    print(my_list)
except ValueError as exc:
    print(f"Caught ValueError: {exc}")
```

```
['Iron man', 'Thor', 'Wonder Woman', 'Wolverine', 'Naruto']
```

`insert`

```
# Insert a object at a particular index.
# We use insert method of the list.
my_list.insert(1, "Super Man")
print(my_list)
```

```
['Iron man', 'Super Man', 'Thor', 'Wonder Woman', 'Wolverine', 'Naruto']
```

`reverse`

```
# Reverse the objects in the list.
# we use reverse method of the list.
my_list.reverse()
print(my_list)

# revert to the actual order
my_list.reverse()

# We have one more method too for this 🙃
# The indexing of the list would be in the form of list[start: end: step]
# We will use step as -1 to get the elements in reverse order 😉
print(my_list[::-1])
```

```
['Naruto', 'Wolverine', 'Wonder Woman', 'Thor', 'Super Man', 'Iron man']
['Naruto', 'Wolverine', 'Wonder Woman', 'Thor', 'Super Man', 'Iron man']
```

`index`

```
# Index of an object in the list.
# we use index method of the list.
# raises a ValueError, if no given object is found in the list.
try:
    print(my_list.index("Naruto"))
except ValueError as exc:
    print(f"Caught ValueError: {exc}")
```

```
5
```

`pop`

```
# Pop is used to remove and return the element present at the last in the
list(index=-1) by default.
# When index argument is passed, it would remove and return the element at that index.
# raises IndexError when no object is present at the given Index.
try:
    last_element = (
        my_list.pop()
    )  # can be passed index argument value, if required to pop at a specific index.
    print(last_element)
except IndexError as exc:
    print(f"Caught IndexError: {exc}")
```

```
Naruto
```

5.3. Set

A set is collection of unique items, the items does not follow insertion order.

Defining an set is pretty similar to a list or tuple, it is enclosed in `{}`

PS 🔔: If we need to have a empty set, `{}` won't create a set, it creates a empty dictionary instead. So we need to create a empty set by using `set()`

```
anime = {"Dragon ball", "One Piece", "Death Note", "Full Metal Alchemist", "Naruto"}
print(anime)
```

```
{'Death Note', 'One Piece', 'Dragon ball', 'Naruto', 'Full Metal Alchemist'}
```

`add`

```
anime.add("Tokyo Ghoul")
print(anime)
```

```
{'Death Note', 'Tokyo Ghoul', 'One Piece', 'Dragon ball', 'Naruto', 'Full Metal
Alchemist'}
```

`remove`

remove method of set can be used to remove a particular object from the set, if the object is not present, KeyError would be raised.

```
try:
    anime.remove("Tokyo Ghoul")
    print(anime)
except KeyError as exc:
    print(
        f"Caught KeyError as there's given anime series present in the anime set:
{exc}"
    )
```

```
{'Death Note', 'One Piece', 'Dragon ball', 'Naruto', 'Full Metal Alchemist'}
```

5.4. Dictionary

As in few other languages, we have hashmaps, Dictionaries in python are similar. It has unique Key - Value pairs.

The Key and Value can be of any object. Each Key-Value pair is separated by a `,`

```
anime_protagonist = {
    "Dragon Ball": "Goku",
    "One Piece": "Luffy",
    "Death Note": "Yagami Light",
    "Full Metal Alchemist": "Edward Elric",
    "Naruto": "Naruto",
}
print(anime_protagonist)
```

```
{'Dragon Ball': 'Goku', 'One Piece': 'Luffy', 'Death Note': 'Yagami Light', 'Full
Metal Alchemist': 'Edward Elric', 'Naruto': 'Naruto'}
```

We can access the values of the dictionary by `<dictionary>[<key>]`. If there's no `<key>` in the dictionary, we would be seeing an KeyError 🔑❌

```
try:
    print(anime_protagonist["Dragon Ball"])
except KeyError as exc:
    print(
        f"👻 Ouch, Keyerror has been raised as no given key is found in the
dictionary: {exc}"
    )
```

```
Goku
```

Iterate over keys, values and both in the dictionary 🐇

```
# Keys
print("===Keys===")
for my_key in anime_protagonist.keys():
    print(my_key)

# Values
print("===Values===")
for my_value in anime_protagonist.values():
    print(my_value)

# Key-Values
print("===Key-Values===")
for my_key, my_value in anime_protagonist.items():
    print(f"{my_key} : {my_value}")
```

```
===Keys===
Dragon Ball
One Piece
Death Note
Full Metal Alchemist
Naruto
===Values===
Goku
Luffy
Yagami Light
Edward Elric
Naruto
===Key-Values===
Dragon Ball : Goku
One Piece : Luffy
Death Note : Yagami Light
Full Metal Alchemist : Edward Elric
Naruto : Naruto
```

PS 🔔: Are dictionaries ordered collection🤔?

From Python 3.7 dictionaries follow insertion order 😎

In python versions older than 3.7, the insertion of items is not ordered🙄. No problem 🙃, we still have OrderedDict(present in collections module) `from collections import OrderedDict` which does the same 😉

6. IDEs/Editors for Python

We have a lot of IDEs/Editors available for Python. Although we get **IDLE** abrevated as **I**ntegrated **D**evelopment and **L**earning **E**nvironment

IDLE gets installed automatically on Windows along with Python installation. On Mac or *nix operating systems we need install it manually

A few great IDEs/Editors for Python

6.1. PyCharm

6.2. Spyder

6.3. Visual Studio Code

6.4. Atom

6.5. Jupyter

6.6. Google Colab

This is my Personal Favourite when I need huge memory and GPU. We get those for free here 😎

PS 😉: I always say to prefer using basic text editor like notepad/gedit when learning a new language and use a good IDE if your Boss wants you to do the work quick 😜

7. User Input

`input` is a builtin function in Python, which prompts for the user to enter as standard input upto newline(`\n`).

`input` function always returns a string datatype, we need to typecast to respective datatype required.

Python 2.x's `input` is different from Python 3.x's `input`.

Python 2.x's `input` evaluates the string as a python command, like `eval(input())`.

```
user_entered = input("Hey Pythonist! Please enter anything: \n>>>")
print(f"The input entered is {user_entered}")
```

```
Hello Python!
The input entered is Hello Python!
```

Let's try typecasting to integers we got from the user.

If the input is not a valid integer value, typecasting to integer raises `ValueError`

```
try:
    variable_1 = input("Enter variable 1 to be added: \n>>>")  # string
    variable_2 = input("Enter variable 2 to be added \n>>>")  # string
    integer_1 = int(variable_1)  # Typecasting to integer
    integer_2 = int(variable_2)  # Typecasting to integer
    print(f"sum of {variable_1} and {variable_2} = {integer_1+integer_2}")
except ValueError as exc:
    print(f"👻 unable to typecast to integer: {exc}")
```

`output`

```
>>>I am a string 😈
>>>I am also a string 😈
👻 unable to typecast to integer: invalid literal for int() with base 10: 'I am a string 😈'
```

8. Builtins

```
import builtins
```

We can see what all builtins does Python provide.

For our sake, we are traversing the complete list and printing the number and builtin attribute.

The function we are usign to traverse in `dir(builtins)` and get index and builtin attribute is `enumerate` which is also a bulitin 😉

```
for index, builtin_attribute in enumerate(dir(builtins)):
    print(f"{index}) {builtin_attribute}")
```

```
0) ArithmeticError
1) AssertionError
2) AttributeError
3) BaseException
4) BlockingIOError
5) BrokenPipeError
6) BufferError
7) BytesWarning
8) ChildProcessError
9) ConnectionAbortedError
10) ConnectionError
11) ConnectionRefusedError
12) ConnectionResetError
13) DeprecationWarning
14) EOFError
15) Ellipsis
16) EnvironmentError
17) Exception
18) False
19) FileExistsError
20) FileNotFoundError
21) FloatingPointError
22) FutureWarning
23) GeneratorExit
24) IOError
25) ImportError
26) ImportWarning
27) IndentationError
28) IndexError
29) InterruptedError
30) IsADirectoryError
31) KeyError
32) KeyboardInterrupt
33) LookupError
34) MemoryError
35) ModuleNotFoundError
36) NameError
37) None
38) NotADirectoryError
39) NotImplemented
40) NotImplementedError
41) OSError
42) OverflowError
43) PendingDeprecationWarning
44) PermissionError
45) ProcessLookupError
46) RecursionError
47) ReferenceError
48) ResourceWarning
49) RuntimeError
50) RuntimeWarning
51) StopAsyncIteration
52) StopIteration
53) SyntaxError
54) SyntaxWarning
55) SystemError
56) SystemExit
57) TabError
58) TimeoutError
59) True
60) TypeError
61) UnboundLocalError
62) UnicodeDecodeError
63) UnicodeEncodeError
64) UnicodeError
65) UnicodeTranslateError
66) UnicodeWarning
67) UserWarning
68) ValueError
69) Warning
70) ZeroDivisionError
71) __IPYTHON__
72) __build_class__
73) __debug__
74) __doc__
75) __import__
76) __loader__
77) __name__
78) __package__
79) __spec__
80) abs
81) all
```

```
82) any
83) ascii
84) bin
85) bool
86) breakpoint
87) bytearray
88) bytes
89) callable
90) chr
91) classmethod
92) compile
93) complex
94) copyright
95) credits
96) delattr
97) dict
98) dir
99) display
100) divmod
101) enumerate
102) eval
103) exec
104) filter
105) float
106) format
107) frozenset
108) get_ipython
109) getattr
110) globals
111) hasattr
112) hash
113) help
114) hex
115) id
116) input
117) int
118) isinstance
119) issubclass
120) iter
121) len
122) license
123) list
124) locals
125) map
126) max
127) memoryview
128) min
129) next
130) object
131) oct
132) open
133) ord
134) pow
135) print
136) property
137) range
138) repr
139) reversed
140) round
141) set
142) setattr
143) slice
144) sorted
145) staticmethod
146) str
147) sum
148) super
149) tuple
150) type
151) vars
152) zip
```

There's a difference between **Keywords** and **Builtins** 🤔. We can't assign a new object to the Keywords, if we try to do, we would be seeing an exception raised 🔴. But coming to builtins, we can assign any object to the builtin names, and Python won't have any issues, but it's not a good practice to do so 😇

9. Module

A module is a importable python file and can be created by creating a file with extension as `.py`

We can import the objects present in the module.

In the below 👇 example, we are importing `hello` function from `greet` module (greet.py)

`greet.py`

```
"""Module to greet the user"""

import getpass


def hello():
    username: str = getpass.getuser().capitalize()
    print(f"Hello {username}. Have a great day :)")


if __name__ == "__main__":
    hello()
```

```
from greet import hello
```

```
hello()
```

```
Hello Naveen. Have a great day :)
```

let's have a look at the greet.py module. Well, we see the below `if` condition.

```
if __name__ == "__main__":
    hello()
```

But why do we we need to have it🤔? We can just call the `hello` function at the end as

```
hello()
```

Let's see the below👇 code to know why we use the first approach rather than the second.🙃

```
import greet
```

🔍 The above code doesn't greet you 😢

```
%run ./greet.py
```

```
Hello Naveen. Have a great day :)
```

But, this above code greets you😎.

The reason for this is, in the first snippet, we are importing a module called `greet`, so the actual code we are executing is in this REPL or Ipython shell.

Coming to second snippet, we are executing the `greet.py` directly.

Value of `__name__` would be "__main__" if we are executing a Python module directly. If we import a module(using the module indirectly) then value of `__name__` would be the relative path of the imported module. In the first example the `__name__` in the greet module would be "greet". As the "greet" is not equal to "__main__", that's the reason, we never went to the `if` condition when we imported greet module. 🙂

10. String representations of objects: str() vs repr()

`str()` and `repr()` are builtin functions used to represent the object in the form of string.

Suppose we have an object `x`.

`str(x)` would be calling the dunder (double underscore) `__str__` method of `x` as `x.__str__()`

`repr(x)` would be calling the dunder (double underscore) `__repr__` method of `x` as `x.__repr__()`

😑 Well, what all are these new terms `__str__` and `__repr__` 🤔?

As we know that Python is object oriented language, and so supports inheritance. In Python, all the classes would inherit from the base class `object`. `object` class has the methods `__str__`, `__repr__` and a lot more (which can be deepdived in someother notebook 😉). Hence every class would be having `__str__` and `__repr__` implicitly 😊

Python's official documentations states that `__str__` should be used to represent a object which is human readable(informal), whereas `__repr__` is used for official representation of an object.

```
from datetime import datetime

now = datetime.now()

print(f"The repr of now is: {repr(now)}")
print(f"The str of now is: {str(now)}")
```

```
The repr of now is: datetime.datetime(2021, 8, 24, 14, 59, 52, 243710)
The str of now is: 2021-08-24 14:59:52.243710
```

```
class ProgrammingLanguage:
    def __init__(self, language: str):
        self.language = language


language_obj = ProgrammingLanguage(language="Python")
print(f"The repr of language_obj is: {repr(language_obj)}")
print(f"The str of language_obj is: {str(language_obj)}")
```

```
The repr of language_obj is: <__main__.ProgrammingLanguage object at 0x7f5d3f1e0e50>
The str of language_obj is: <__main__.ProgrammingLanguage object at 0x7f5d3f1e0e50>
```

In the above example we see that default repr output. The address of the object might be different for everyone.

Now let's try to override the `__str__` and `__repr__` methods and see how the representations work

```
class Human:
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    # overriding __str__ method
    def __str__(self):
        return f"I am {self.name} of age {self.age}"

    # overriding __repr__ method
    def __repr__(self):
        return f"Human(name={self.name}, age={self.age}) object at {hex(id(self))}"


human_obj = Human(name="IronMan", age=48)
print(f"The repr of human_obj is: {repr(human_obj)}")
print(f"The str of human_obj is: {str(human_obj)}")
```

```
The repr of human_obj is: Human(name=IronMan, age=48) object at 0x7f5d3c1290d0
The str of human_obj is: I am IronMan of age 48
```

We see that the result representations of the `human_obj` have been changed as we have overridden the `__str__` and `__repr__` methods 😊

11. Installing packages

Python has one of the largest programming community who build 3rd party packages and support community help ❤️.

That's pretty good, Now, how do we install the packages 🤔? We could use Python's package manager **PIP**.

Python's official 3rd party package repository is Python Package Index (PyPI) and its index url is https://pypi.org/simple

Here's how to use PIP in shell/terminal:

To search for a package:

```
pip search [package name]
```

To install a package: Install

```
    pip install [package name]
```

Install a specific version

```
    pip install [package name]==[version]
```

Install greater than a specific version

```
    pip install [package name]>=[verion]
```

To uninstall a package

```
pip uninstall [package name]
```

11.1. Tidbits 🔔

There are modern ways of managing the dependencies using poetry, flit etc.. We will get to those soon…😊

12. Help Utility

Python has a builtin help utility which helps to know about the keywords, builtin functions, modules.

```
help()
```

You can pass keyword, bulitin function or Module to help function to know about the same.

```
import os
```

```
# Help utility on the builtin module 'sys'
help(os)
```

```
Help on module os:

NAME
    os - OS routines for NT or Posix depending on what system we're on.

MODULE REFERENCE
    https://docs.python.org/3.9/library/os

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This exports:
      - all functions from posix or nt, e.g. unlink, stat, etc.
      - os.path is either posixpath or ntpath
      - os.name is either 'posix' or 'nt'
      - os.curdir is a string representing the current directory (always '.')
      - os.pardir is a string representing the parent directory (always '..')
      - os.sep is the (or a most common) pathname separator ('/' or '\\')
      - os.extsep is the extension separator (always '.')
      - os.altsep is the alternate pathname separator (None or '/')
      - os.pathsep is the component separator used in $PATH etc
      - os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
      - os.defpath is the default search path for executables
      - os.devnull is the file path of the null device ('/dev/null', etc.)

    Programs that import and use 'os' stand a better chance of being
    portable between different platforms.  Of course, they must then
    only use functions that are defined by all platforms (e.g., unlink
    and opendir), and leave all pathname manipulation to os.path
```

```
    (e.g., split and join).

CLASSES
    builtins.Exception(builtins.BaseException)
        builtins.OSError
    builtins.object
        posix.DirEntry
    builtins.tuple(builtins.object)
        stat_result
        statvfs_result
        terminal_size
        posix.sched_param
        posix.times_result
        posix.uname_result
        posix.waitid_result

    class DirEntry(builtins.object)
     |  Methods defined here:
     |
     |  __fspath__(self, /)
     |      Returns the path for the entry.
     |
     |  __repr__(self, /)
     |      Return repr(self).
     |
     |  inode(self, /)
     |      Return inode of the entry; cached per entry.
     |
     |  is_dir(self, /, *, follow_symlinks=True)
     |      Return True if the entry is a directory; cached per entry.
     |
     |  is_file(self, /, *, follow_symlinks=True)
     |      Return True if the entry is a file; cached per entry.
     |
     |  is_symlink(self, /)
     |      Return True if the entry is a symbolic link; cached per entry.
     |
     |  stat(self, /, *, follow_symlinks=True)
     |      Return stat_result object for the entry; cached per entry.
     |
     |  ----------------------------------------------------------------------
     |  Class methods defined here:
     |
     |  __class_getitem__(...) from builtins.type
     |      See PEP 585
     |
     |  ----------------------------------------------------------------------
     |  Static methods defined here:
     |
     |  __new__(*args, **kwargs) from builtins.type
     |      Create and return a new object.  See help(type) for accurate signature.
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  name
     |      the entry's base filename, relative to scandir() "path" argument
     |
     |  path
     |      the entry's full path name; equivalent to os.path.join(scandir_path,
entry.name)

    error = class OSError(Exception)
     |  Base class for I/O related errors.
     |
     |  Method resolution order:
     |      OSError
     |      Exception
     |      BaseException
     |      object
     |
     |  Built-in subclasses:
     |      BlockingIOError
     |      ChildProcessError
     |      ConnectionError
     |      FileExistsError
     |      ... and 7 other subclasses
     |
     |  Methods defined here:
     |
     |  __init__(self, /, *args, **kwargs)
     |      Initialize self.  See help(type(self)) for accurate signature.
     |
     |  __reduce__(...)
     |      Helper for pickle.
     |
     |  __str__(self, /)
```

```
 |      Return str(self).
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  characters_written
 |
 |  errno
 |      POSIX exception code
 |
 |  filename
 |      exception filename
 |
 |  filename2
 |      second exception filename
 |
 |  strerror
 |      exception strerror
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from BaseException:
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __setattr__(self, name, value, /)
 |      Implement setattr(self, name, value).
 |
 |  __setstate__(...)
 |
 |  with_traceback(...)
 |      Exception.with_traceback(tb) --
 |      set self.__traceback__ to tb and return self.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from BaseException:
 |
 |  __cause__
 |      exception cause
 |
 |  __context__
 |      exception context
 |
 |  __dict__
 |
 |  __suppress_context__
 |
 |  __traceback__
 |
 |  args

class sched_param(builtins.tuple)
 |  sched_param(iterable=(), /)
 |
 |  Currently has only one field: sched_priority
 |
 |  sched_priority
 |    A scheduling parameter.
 |
 |  Method resolution order:
 |      sched_param
 |      builtins.tuple
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __reduce__(...)
 |      Helper for pickle.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
```

```
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  sched_priority
 |      the scheduling priority
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  n_fields = 1
 |
 |  n_sequence_fields = 1
 |
 |  n_unnamed_fields = 0
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.tuple:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __getnewargs__(self, /)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  index(self, value, start=0, stop=9223372036854775807, /)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from builtins.tuple:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585

class stat_result(builtins.tuple)
 |  stat_result(iterable=(), /)
 |
 |  stat_result: Result from stat, fstat, or lstat.
 |
```

```
|  This object may be accessed either as a tuple of
|    (mode, ino, dev, nlink, uid, gid, size, atime, mtime, ctime)
|  or via the attributes st_mode, st_ino, st_dev, st_nlink, st_uid, and so on.
|
|  Posix/windows: If your platform supports st_blksize, st_blocks, st_rdev,
|  or st_flags, they are available as attributes only.
|
|  See os.stat for more information.
|
|  Method resolution order:
|      stat_result
|      builtins.tuple
|      builtins.object
|
|  Methods defined here:
|
|  __reduce__(...)
|      Helper for pickle.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  st_atime
|      time of last access
|
|  st_atime_ns
|      time of last access in nanoseconds
|
|  st_blksize
|      blocksize for filesystem I/O
|
|  st_blocks
|      number of blocks allocated
|
|  st_ctime
|      time of last change
|
|  st_ctime_ns
|      time of last change in nanoseconds
|
|  st_dev
|      device
|
|  st_gid
|      group ID of owner
|
|  st_ino
|      inode
|
|  st_mode
|      protection bits
|
|  st_mtime
|      time of last modification
|
|  st_mtime_ns
|      time of last modification in nanoseconds
|
|  st_nlink
|      number of hard links
|
|  st_rdev
|      device type (if inode device)
|
|  st_size
|      total size, in bytes
|
|  st_uid
|      user ID of owner
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  n_fields = 19
|
|  n_sequence_fields = 10
|
|  n_unnamed_fields = 3
```

```
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.tuple:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __getnewargs__(self, /)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  index(self, value, start=0, stop=9223372036854775807, /)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from builtins.tuple:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585

class statvfs_result(builtins.tuple)
 |  statvfs_result(iterable=(), /)
 |
 |  statvfs_result: Result from statvfs or fstatvfs.
 |
 |  This object may be accessed either as a tuple of
 |    (bsize, frsize, blocks, bfree, bavail, files, ffree, favail, flag,
namemax),
 |  or via the attributes f_bsize, f_frsize, f_blocks, f_bfree, and so on.
 |
 |  See os.statvfs for more information.
 |
 |  Method resolution order:
 |      statvfs_result
 |      builtins.tuple
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __reduce__(...)
 |      Helper for pickle.
 |
```

```
|  __repr__(self, /)
|      Return repr(self).
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  f_bavail
|
|  f_bfree
|
|  f_blocks
|
|  f_bsize
|
|  f_favail
|
|  f_ffree
|
|  f_files
|
|  f_flag
|
|  f_frsize
|
|  f_fsid
|
|  f_namemax
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  n_fields = 11
|
|  n_sequence_fields = 10
|
|  n_unnamed_fields = 0
|
|  ----------------------------------------------------------------------
|  Methods inherited from builtins.tuple:
|
|  __add__(self, value, /)
|      Return self+value.
|
|  __contains__(self, key, /)
|      Return key in self.
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(self, key, /)
|      Return self[key].
|
|  __getnewargs__(self, /)
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __hash__(self, /)
|      Return hash(self).
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
```

```
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  index(self, value, start=0, stop=9223372036854775807, /)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from builtins.tuple:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585

class terminal_size(builtins.tuple)
 |  terminal_size(iterable=(), /)
 |
 |  A tuple of (columns, lines) for holding terminal window size
 |
 |  Method resolution order:
 |      terminal_size
 |      builtins.tuple
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __reduce__(...)
 |      Helper for pickle.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  columns
 |      width of the terminal window in characters
 |
 |  lines
 |      height of the terminal window in characters
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  n_fields = 2
 |
 |  n_sequence_fields = 2
 |
 |  n_unnamed_fields = 0
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.tuple:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __getnewargs__(self, /)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
```

```
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  index(self, value, start=0, stop=9223372036854775807, /)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from builtins.tuple:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585

class times_result(builtins.tuple)
 |  times_result(iterable=(), /)
 |
 |  times_result: Result from os.times().
 |
 |  This object may be accessed either as a tuple of
 |    (user, system, children_user, children_system, elapsed),
 |  or via the attributes user, system, children_user, children_system,
 |  and elapsed.
 |
 |  See os.times for more information.
 |
 |  Method resolution order:
 |      times_result
 |      builtins.tuple
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __reduce__(...)
 |      Helper for pickle.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  children_system
 |      system time of children
 |
 |  children_user
 |      user time of children
 |
 |  elapsed
 |      elapsed time since an arbitrary point in the past
 |
 |  system
 |      system time
 |
 |  user
 |      user time
```

```
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  n_fields = 5
 |
 |  n_sequence_fields = 5
 |
 |  n_unnamed_fields = 0
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.tuple:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __getnewargs__(self, /)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  index(self, value, start=0, stop=9223372036854775807, /)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from builtins.tuple:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585

class uname_result(builtins.tuple)
 |  uname_result(iterable=(), /)
 |
 |  uname_result: Result from os.uname().
 |
 |  This object may be accessed either as a tuple of
 |    (sysname, nodename, release, version, machine),
 |  or via the attributes sysname, nodename, release, version, and machine.
 |
 |  See os.uname for more information.
 |
 |  Method resolution order:
 |      uname_result
```

```
|      builtins.tuple
|      builtins.object
|
|  Methods defined here:
|
|  __reduce__(...)
|      Helper for pickle.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  machine
|      hardware identifier
|
|  nodename
|      name of machine on network (implementation-defined)
|
|  release
|      operating system release
|
|  sysname
|      operating system name
|
|  version
|      operating system version
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  n_fields = 5
|
|  n_sequence_fields = 5
|
|  n_unnamed_fields = 0
|
|  ----------------------------------------------------------------------
|  Methods inherited from builtins.tuple:
|
|  __add__(self, value, /)
|      Return self+value.
|
|  __contains__(self, key, /)
|      Return key in self.
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(self, key, /)
|      Return self[key].
|
|  __getnewargs__(self, /)
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __hash__(self, /)
|      Return hash(self).
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.
```

```
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  index(self, value, start=0, stop=9223372036854775807, /)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from builtins.tuple:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585

class waitid_result(builtins.tuple)
 |  waitid_result(iterable=(), /)
 |
 |  waitid_result: Result from waitid.
 |
 |  This object may be accessed either as a tuple of
 |    (si_pid, si_uid, si_signo, si_status, si_code),
 |  or via the attributes si_pid, si_uid, and so on.
 |
 |  See os.waitid for more information.
 |
 |  Method resolution order:
 |      waitid_result
 |      builtins.tuple
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __reduce__(...)
 |      Helper for pickle.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  si_code
 |
 |  si_pid
 |
 |  si_signo
 |
 |  si_status
 |
 |  si_uid
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  n_fields = 5
 |
 |  n_sequence_fields = 5
 |
 |  n_unnamed_fields = 0
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.tuple:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
```

```
     |
     |  __getattribute__(self, name, /)
     |      Return getattr(self, name).
     |
     |  __getitem__(self, key, /)
     |      Return self[key].
     |
     |  __getnewargs__(self, /)
     |
     |  __gt__(self, value, /)
     |      Return self>value.
     |
     |  __hash__(self, /)
     |      Return hash(self).
     |
     |  __iter__(self, /)
     |      Implement iter(self).
     |
     |  __le__(self, value, /)
     |      Return self<=value.
     |
     |  __len__(self, /)
     |      Return len(self).
     |
     |  __lt__(self, value, /)
     |      Return self<value.
     |
     |  __mul__(self, value, /)
     |      Return self*value.
     |
     |  __ne__(self, value, /)
     |      Return self!=value.
     |
     |  __rmul__(self, value, /)
     |      Return value*self.
     |
     |  count(self, value, /)
     |      Return number of occurrences of value.
     |
     |  index(self, value, start=0, stop=9223372036854775807, /)
     |      Return first index of value.
     |
     |      Raises ValueError if the value is not present.
     |
     |  ----------------------------------------------------------------------
     |  Class methods inherited from builtins.tuple:
     |
     |  __class_getitem__(...) from builtins.type
     |      See PEP 585

FUNCTIONS
    WCOREDUMP(status, /)
        Return True if the process returning status was dumped to a core file.

    WEXITSTATUS(status)
        Return the process return code from status.

    WIFCONTINUED(status)
        Return True if a particular process was continued from a job control stop.

        Return True if the process returning status was continued from a
        job control stop.

    WIFEXITED(status)
        Return True if the process returning status exited via the exit() system
call.

    WIFSIGNALED(status)
        Return True if the process returning status was terminated by a signal.

    WIFSTOPPED(status)
        Return True if the process returning status was stopped.

    WSTOPSIG(status)
        Return the signal that stopped the process that provided the status value.

    WTERMSIG(status)
        Return the signal that terminated the process that provided the status value.

    _exit(status)
        Exit to the system with specified status, without normal exit processing.

    abort()
        Abort the interpreter immediately.

        This function 'dumps core' or otherwise fails in the hardest way possible
        on the hosting operating system.  This function never returns.
```

```
    access(path, mode, *, dir_fd=None, effective_ids=False, follow_symlinks=True)
        Use the real uid/gid to test for access to a path.

          path
            Path to be tested; can be string, bytes, or a path-like object.
          mode
            Operating-system mode bitfield.  Can be F_OK to test existence,
            or the inclusive-OR of R_OK, W_OK, and X_OK.
          dir_fd
            If not None, it should be a file descriptor open to a directory,
            and path should be relative; path will then be relative to that
            directory.
          effective_ids
            If True, access will use the effective uid/gid instead of
            the real uid/gid.
          follow_symlinks
            If False, and the last element of the path is a symbolic link,
            access will examine the symbolic link itself instead of the file
            the link points to.

        dir_fd, effective_ids, and follow_symlinks may not be implemented
          on your platform.  If they are unavailable, using them will raise a
          NotImplementedError.

        Note that most operations will use the effective uid/gid, therefore this
          routine can be used in a suid/sgid environment to test if the invoking user
          has the specified access to the path.

    chdir(path)
        Change the current working directory to the specified path.

        path may always be specified as a string.
        On some platforms, path may also be specified as an open file descriptor.
          If this functionality is unavailable, using it raises an exception.

    chmod(path, mode, *, dir_fd=None, follow_symlinks=True)
        Change the access permissions of a file.

          path
            Path to be modified.  May always be specified as a str, bytes, or a path-
like object.
            On some platforms, path may also be specified as an open file descriptor.
            If this functionality is unavailable, using it raises an exception.
          mode
            Operating-system mode bitfield.
          dir_fd
            If not None, it should be a file descriptor open to a directory,
            and path should be relative; path will then be relative to that
            directory.
          follow_symlinks
            If False, and the last element of the path is a symbolic link,
            chmod will modify the symbolic link itself instead of the file
            the link points to.

        It is an error to use dir_fd or follow_symlinks when specifying path as
          an open file descriptor.
        dir_fd and follow_symlinks may not be implemented on your platform.
          If they are unavailable, using them will raise a NotImplementedError.

    chown(path, uid, gid, *, dir_fd=None, follow_symlinks=True)
        Change the owner and group id of path to the numeric uid and gid.\

          path
            Path to be examined; can be string, bytes, a path-like object, or open-
file-descriptor int.
          dir_fd
            If not None, it should be a file descriptor open to a directory,
            and path should be relative; path will then be relative to that
            directory.
          follow_symlinks
            If False, and the last element of the path is a symbolic link,
            stat will examine the symbolic link itself instead of the file
            the link points to.

        path may always be specified as a string.
        On some platforms, path may also be specified as an open file descriptor.
          If this functionality is unavailable, using it raises an exception.
        If dir_fd is not None, it should be a file descriptor open to a directory,
          and path should be relative; path will then be relative to that directory.
        If follow_symlinks is False, and the last element of the path is a symbolic
          link, chown will modify the symbolic link itself instead of the file the
          link points to.
        It is an error to use dir_fd or follow_symlinks when specifying path as
          an open file descriptor.
        dir_fd and follow_symlinks may not be implemented on your platform.
          If they are unavailable, using them will raise a NotImplementedError.
```

```
chroot(path)
    Change root directory to path.

close(fd)
    Close a file descriptor.

closerange(fd_low, fd_high, /)
    Closes all file descriptors in [fd_low, fd_high), ignoring errors.

confstr(name, /)
    Return a string-valued system configuration variable.

copy_file_range(src, dst, count, offset_src=None, offset_dst=None)
    Copy count bytes from one file descriptor to another.

      src
        Source file descriptor.
      dst
        Destination file descriptor.
      count
        Number of bytes to copy.
      offset_src
        Starting offset in src.
      offset_dst
        Starting offset in dst.

    If offset_src is None, then src is read from the current position;
    respectively for offset_dst.

cpu_count()
    Return the number of CPUs in the system; return None if indeterminable.

    This number is not equivalent to the number of CPUs the current process can
    use.  The number of usable CPUs can be obtained with
    ``len(os.sched_getaffinity(0))``

ctermid()
    Return the name of the controlling terminal for this process.

device_encoding(fd)
    Return a string describing the encoding of a terminal's file descriptor.

    The file descriptor must be attached to a terminal.
    If the device is not a terminal, return None.

dup(fd, /)
    Return a duplicate of a file descriptor.

dup2(fd, fd2, inheritable=True)
    Duplicate file descriptor.

execl(file, *args)
    execl(file, *args)

    Execute the executable file with argument list args, replacing the
    current process.

execle(file, *args)
    execle(file, *args, env)

    Execute the executable file with argument list args and
    environment env, replacing the current process.

execlp(file, *args)
    execlp(file, *args)

    Execute the executable file (which is searched for along $PATH)
    with argument list args, replacing the current process.

execlpe(file, *args)
    execlpe(file, *args, env)

    Execute the executable file (which is searched for along $PATH)
    with argument list args and environment env, replacing the current
    process.

execv(path, argv, /)
    Execute an executable path with arguments, replacing current process.

    path
      Path of executable file.
    argv
      Tuple or list of strings.

execve(path, argv, env)
    Execute an executable path with arguments, replacing current process.
```

```
    path
      Path of executable file.
    argv
      Tuple or list of strings.
    env
      Dictionary of strings mapping to strings.

execvp(file, args)
    execvp(file, args)

    Execute the executable file (which is searched for along $PATH)
    with argument list args, replacing the current process.
    args may be a list or tuple of strings.

execvpe(file, args, env)
    execvpe(file, args, env)

    Execute the executable file (which is searched for along $PATH)
    with argument list args and environment env, replacing the
    current process.
    args may be a list or tuple of strings.

fchdir(fd)
    Change to the directory of the given file descriptor.

    fd must be opened on a directory, not a file.
    Equivalent to os.chdir(fd).

fchmod(fd, mode)
    Change the access permissions of the file given by file descriptor fd.

    Equivalent to os.chmod(fd, mode).

fchown(fd, uid, gid)
    Change the owner and group id of the file specified by file descriptor.

    Equivalent to os.chown(fd, uid, gid).

fdatasync(fd)
    Force write of fd to disk without forcing update of metadata.

fdopen(fd, *args, **kwargs)
    # Supply os.fdopen()

fork()
    Fork a child process.

    Return 0 to child process and PID of child to parent process.

forkpty()
    Fork a new process with a new pseudo-terminal as controlling tty.

    Returns a tuple of (pid, master_fd).
    Like fork(), return pid of 0 to the child process,
    and pid of child to the parent process.
    To both, return fd of newly opened pseudo-terminal.

fpathconf(fd, name, /)
    Return the configuration limit name for the file descriptor fd.

    If there is no limit, return -1.

fsdecode(filename)
    Decode filename (an os.PathLike, bytes, or str) from the filesystem
    encoding with 'surrogateescape' error handler, return str unchanged. On
    Windows, use 'strict' error handler if the file system encoding is
    'mbcs' (which is the default encoding).

fsencode(filename)
    Encode filename (an os.PathLike, bytes, or str) to the filesystem
    encoding with 'surrogateescape' error handler, return bytes unchanged.
    On Windows, use 'strict' error handler if the file system encoding is
    'mbcs' (which is the default encoding).

fspath(path)
    Return the file system path representation of the object.

    If the object is str or bytes, then allow it to pass through as-is. If the
    object defines __fspath__(), then return the result of that method. All other
    types raise a TypeError.

fstat(fd)
    Perform a stat system call on the given file descriptor.

    Like stat(), but for an open file descriptor.
    Equivalent to os.stat(fd).
```

```
fstatvfs(fd, /)
    Perform an fstatvfs system call on the given fd.

    Equivalent to statvfs(fd).

fsync(fd)
    Force write of fd to disk.

ftruncate(fd, length, /)
    Truncate a file, specified by file descriptor, to a specific length.

fwalk(top='.', topdown=True, onerror=None, *, follow_symlinks=False, dir_fd=None)
    Directory tree generator.

    This behaves exactly like walk(), except that it yields a 4-tuple

        dirpath, dirnames, filenames, dirfd

    `dirpath`, `dirnames` and `filenames` are identical to walk() output,
    and `dirfd` is a file descriptor referring to the directory `dirpath`.

    The advantage of fwalk() over walk() is that it's safe against symlink
    races (when follow_symlinks is False).

    If dir_fd is not None, it should be a file descriptor open to a directory,
      and top should be relative; top will then be relative to that directory.
      (dir_fd is always supported for fwalk.)

    Caution:
    Since fwalk() yields file descriptors, those are only valid until the
    next iteration step, so you should dup() them if you want to keep them
    for a longer period.

    Example:

    import os
    for root, dirs, files, rootfd in os.fwalk('python/Lib/email'):
        print(root, "consumes", end="")
        print(sum(os.stat(name, dir_fd=rootfd).st_size for name in files),
              end="")
        print("bytes in", len(files), "non-directory files")
        if 'CVS' in dirs:
            dirs.remove('CVS')  # don't visit CVS directories

get_blocking(fd, /)
    Get the blocking mode of the file descriptor.

    Return False if the O_NONBLOCK flag is set, True if the flag is cleared.

get_exec_path(env=None)
    Returns the sequence of directories that will be searched for the
    named executable (similar to a shell) when launching a process.

    *env* must be an environment variable dict or None.  If *env* is None,
    os.environ will be used.

get_inheritable(fd, /)
    Get the close-on-exe flag of the specified file descriptor.

get_terminal_size(...)
    Return the size of the terminal window as (columns, lines).

    The optional argument fd (default standard output) specifies
    which file descriptor should be queried.

    If the file descriptor is not connected to a terminal, an OSError
    is thrown.

    This function will only be defined if an implementation is
    available for this system.

    shutil.get_terminal_size is the high-level function which should
    normally be used, os.get_terminal_size is the low-level implementation.

getcwd()
    Return a unicode string representing the current working directory.

getcwdb()
    Return a bytes string representing the current working directory.

getegid()
    Return the current process's effective group id.

getenv(key, default=None)
    Get an environment variable, return None if it doesn't exist.
    The optional second argument can specify an alternate default.
```

```
    key, default and the result are str.

getenvb(key, default=None)
    Get an environment variable, return None if it doesn't exist.
    The optional second argument can specify an alternate default.
    key, default and the result are bytes.

geteuid()
    Return the current process's effective user id.

getgid()
    Return the current process's group id.

getgrouplist(user, group, /)
    Returns a list of groups to which a user belongs.

    user
      username to lookup
    group
      base group id of the user

getgroups()
    Return list of supplemental group IDs for the process.

getloadavg()
    Return average recent system load information.

    Return the number of processes in the system run queue averaged over
    the last 1, 5, and 15 minutes as a tuple of three floats.
    Raises OSError if the load average was unobtainable.

getlogin()
    Return the actual login name.

getpgid(pid)
    Call the system call getpgid(), and return the result.

getpgrp()
    Return the current process group id.

getpid()
    Return the current process id.

getppid()
    Return the parent's process id.

    If the parent process has already exited, Windows machines will still
    return its id; others systems will return the id of the 'init' process (1).

getpriority(which, who)
    Return program scheduling priority.

getrandom(size, flags=0)
    Obtain a series of random bytes.

getresgid()
    Return a tuple of the current process's real, effective, and saved group ids.

getresuid()
    Return a tuple of the current process's real, effective, and saved user ids.

getsid(pid, /)
    Call the system call getsid(pid) and return the result.

getuid()
    Return the current process's user id.

getxattr(path, attribute, *, follow_symlinks=True)
    Return the value of extended attribute attribute on path.

    path may be either a string, a path-like object, or an open file descriptor.
    If follow_symlinks is False, and the last element of the path is a symbolic
      link, getxattr will examine the symbolic link itself instead of the file
      the link points to.

initgroups(username, gid, /)
    Initialize the group access list.

    Call the system initgroups() to initialize the group access list with all of
    the groups of which the specified username is a member, plus the specified
    group id.

isatty(fd, /)
    Return True if the fd is connected to a terminal.

    Return True if the file descriptor is an open file descriptor
    connected to the slave end of a terminal.
```

```
    kill(pid, signal, /)
        Kill a process with a signal.

    killpg(pgid, signal, /)
        Kill a process group with a signal.

    lchown(path, uid, gid)
        Change the owner and group id of path to the numeric uid and gid.

        This function will not follow symbolic links.
        Equivalent to os.chown(path, uid, gid, follow_symlinks=False).

    link(src, dst, *, src_dir_fd=None, dst_dir_fd=None, follow_symlinks=True)
        Create a hard link to a file.

        If either src_dir_fd or dst_dir_fd is not None, it should be a file
          descriptor open to a directory, and the respective path string (src or dst)
          should be relative; the path will then be relative to that directory.
        If follow_symlinks is False, and the last element of src is a symbolic
          link, link will create a link to the symbolic link itself instead of the
          file the link points to.
        src_dir_fd, dst_dir_fd, and follow_symlinks may not be implemented on your
          platform.  If they are unavailable, using them will raise a
          NotImplementedError.

    listdir(path=None)
        Return a list containing the names of the files in the directory.

        path can be specified as either str, bytes, or a path-like object.  If path
is bytes,
          the filenames returned will also be bytes; in all other circumstances
          the filenames returned will be str.
        If path is None, uses the path='.'.
        On some platforms, path may also be specified as an open file descriptor;\
          the file descriptor must refer to a directory.
          If this functionality is unavailable, using it raises NotImplementedError.

        The list is in arbitrary order.  It does not include the special
        entries '.' and '..' even if they are present in the directory.

    listxattr(path=None, *, follow_symlinks=True)
        Return a list of extended attributes on path.

        path may be either None, a string, a path-like object, or an open file
descriptor.
        if path is None, listxattr will examine the current directory.
        If follow_symlinks is False, and the last element of the path is a symbolic
          link, listxattr will examine the symbolic link itself instead of the file
          the link points to.

    lockf(fd, command, length, /)
        Apply, test or remove a POSIX lock on an open file descriptor.

        fd
          An open file descriptor.
        command
          One of F_LOCK, F_TLOCK, F_ULOCK or F_TEST.
        length
          The number of bytes to lock, starting at the current position.

    lseek(fd, position, how, /)
        Set the position of a file descriptor.  Return the new position.

        Return the new cursor position in number of bytes
        relative to the beginning of the file.

    lstat(path, *, dir_fd=None)
        Perform a stat system call on the given path, without following symbolic
links.

        Like stat(), but do not follow symbolic links.
        Equivalent to stat(path, follow_symlinks=False).

    major(device, /)
        Extracts a device major number from a raw device number.

    makedev(major, minor, /)
        Composes a raw device number from the major and minor device numbers.

    makedirs(name, mode=511, exist_ok=False)
        makedirs(name [, mode=0o777][, exist_ok=False])

        Super-mkdir; create a leaf directory and all intermediate ones.  Works like
        mkdir, except that any intermediate path segment (not just the rightmost)
        will be created if it does not exist. If the target directory already
        exists, raise an OSError if exist_ok is False. Otherwise no exception is
```

```
        raised.  This is recursive.

    memfd_create(name, flags=1)

    minor(device, /)
        Extracts a device minor number from a raw device number.

    mkdir(path, mode=511, *, dir_fd=None)
        Create a directory.

        If dir_fd is not None, it should be a file descriptor open to a directory,
          and path should be relative; path will then be relative to that directory.
        dir_fd may not be implemented on your platform.
          If it is unavailable, using it will raise a NotImplementedError.

        The mode argument is ignored on Windows.

    mkfifo(path, mode=438, *, dir_fd=None)
        Create a "fifo" (a POSIX named pipe).

        If dir_fd is not None, it should be a file descriptor open to a directory,
          and path should be relative; path will then be relative to that directory.
        dir_fd may not be implemented on your platform.
          If it is unavailable, using it will raise a NotImplementedError.

    mknod(path, mode=384, device=0, *, dir_fd=None)
        Create a node in the file system.

        Create a node in the file system (file, device special file or named pipe)
        at path.  mode specifies both the permissions to use and the
        type of node to be created, being combined (bitwise OR) with one of
        S_IFREG, S_IFCHR, S_IFBLK, and S_IFIFO.  If S_IFCHR or S_IFBLK is set on
mode,
        device defines the newly created device special file (probably using
        os.makedev()).  Otherwise device is ignored.

        If dir_fd is not None, it should be a file descriptor open to a directory,
          and path should be relative; path will then be relative to that directory.
        dir_fd may not be implemented on your platform.
          If it is unavailable, using it will raise a NotImplementedError.

    nice(increment, /)
        Add increment to the priority of process and return the new priority.

    open(path, flags, mode=511, *, dir_fd=None)
        Open a file for low level IO.  Returns a file descriptor (integer).

        If dir_fd is not None, it should be a file descriptor open to a directory,
          and path should be relative; path will then be relative to that directory.
        dir_fd may not be implemented on your platform.
          If it is unavailable, using it will raise a NotImplementedError.

    openpty()
        Open a pseudo-terminal.

        Return a tuple of (master_fd, slave_fd) containing open file descriptors
        for both the master and slave ends.

    pathconf(path, name)
        Return the configuration limit name for the file or directory path.

        If there is no limit, return -1.
        On some platforms, path may also be specified as an open file descriptor.
          If this functionality is unavailable, using it raises an exception.

    pidfd_open(pid, flags=0)
        Return a file descriptor referring to the process *pid*.

        The descriptor can be used to perform process management without races and
        signals.

    pipe()
        Create a pipe.

        Returns a tuple of two file descriptors:
          (read_fd, write_fd)

    pipe2(flags, /)
        Create a pipe with flags set atomically.

        Returns a tuple of two file descriptors:
          (read_fd, write_fd)

        flags can be constructed by ORing together one or more of these values:
        O_NONBLOCK, O_CLOEXEC.

    popen(cmd, mode='r', buffering=-1)
```

```
        # Supply os.popen()

    posix_fadvise(fd, offset, length, advice, /)
        Announce an intention to access data in a specific pattern.

        Announce an intention to access data in a specific pattern, thus allowing
        the kernel to make optimizations.
        The advice applies to the region of the file specified by fd starting at
        offset and continuing for length bytes.
        advice is one of POSIX_FADV_NORMAL, POSIX_FADV_SEQUENTIAL,
        POSIX_FADV_RANDOM, POSIX_FADV_NOREUSE, POSIX_FADV_WILLNEED, or
        POSIX_FADV_DONTNEED.

    posix_fallocate(fd, offset, length, /)
        Ensure a file has allocated at least a particular number of bytes on disk.

        Ensure that the file specified by fd encompasses a range of bytes
        starting at offset bytes from the beginning and continuing for length bytes.

    posix_spawn(...)
        Execute the program specified by path in a new process.

        path
          Path of executable file.
        argv
          Tuple or list of strings.
        env
          Dictionary of strings mapping to strings.
        file_actions
          A sequence of file action tuples.
        setpgroup
          The pgroup to use with the POSIX_SPAWN_SETPGROUP flag.
        resetids
          If the value is `true` the POSIX_SPAWN_RESETIDS will be activated.
        setsid
          If the value is `true` the POSIX_SPAWN_SETSID or POSIX_SPAWN_SETSID_NP will
be activated.
        setsigmask
          The sigmask to use with the POSIX_SPAWN_SETSIGMASK flag.
        setsigdef
          The sigmask to use with the POSIX_SPAWN_SETSIGDEF flag.
        scheduler
          A tuple with the scheduler policy (optional) and parameters.

    posix_spawnp(...)
        Execute the program specified by path in a new process.

        path
          Path of executable file.
        argv
          Tuple or list of strings.
        env
          Dictionary of strings mapping to strings.
        file_actions
          A sequence of file action tuples.
        setpgroup
          The pgroup to use with the POSIX_SPAWN_SETPGROUP flag.
        resetids
          If the value is `True` the POSIX_SPAWN_RESETIDS will be activated.
        setsid
          If the value is `True` the POSIX_SPAWN_SETSID or POSIX_SPAWN_SETSID_NP will
be activated.
        setsigmask
          The sigmask to use with the POSIX_SPAWN_SETSIGMASK flag.
        setsigdef
          The sigmask to use with the POSIX_SPAWN_SETSIGDEF flag.
        scheduler
          A tuple with the scheduler policy (optional) and parameters.

    pread(fd, length, offset, /)
        Read a number of bytes from a file descriptor starting at a particular
offset.

        Read length bytes from file descriptor fd, starting at offset bytes from
        the beginning of the file.  The file offset remains unchanged.

    preadv(fd, buffers, offset, flags=0, /)
        Reads from a file descriptor into a number of mutable bytes-like objects.

        Combines the functionality of readv() and pread(). As readv(), it will
        transfer data into each buffer until it is full and then move on to the next
        buffer in the sequence to hold the rest of the data. Its fourth argument,
        specifies the file offset at which the input operation is to be performed. It
        will return the total number of bytes read (which can be less than the total
        capacity of all the objects).

        The flags argument contains a bitwise OR of zero or more of the following
```

```
flags:

        - RWF_HIPRI
        - RWF_NOWAIT

        Using non-zero flags requires Linux 4.6 or newer.

    putenv(name, value, /)
        Change or add an environment variable.

    pwrite(fd, buffer, offset, /)
        Write bytes to a file descriptor starting at a particular offset.

        Write buffer to fd, starting at offset bytes from the beginning of
        the file.  Returns the number of bytes writte.  Does not change the
        current file offset.

    pwritev(fd, buffers, offset, flags=0, /)
        Writes the contents of bytes-like objects to a file descriptor at a given
offset.

        Combines the functionality of writev() and pwrite(). All buffers must be a
sequence
        of bytes-like objects. Buffers are processed in array order. Entire contents
of first
        buffer is written before proceeding to second, and so on. The operating
system may
        set a limit (sysconf() value SC_IOV_MAX) on the number of buffers that can be
used.
        This function writes the contents of each object to the file descriptor and
returns
        the total number of bytes written.

        The flags argument contains a bitwise OR of zero or more of the following
flags:

        - RWF_DSYNC
        - RWF_SYNC

        Using non-zero flags requires Linux 4.7 or newer.

    read(fd, length, /)
        Read from a file descriptor.  Returns a bytes object.

    readlink(path, *, dir_fd=None)
        Return a string representing the path to which the symbolic link points.

        If dir_fd is not None, it should be a file descriptor open to a directory,
        and path should be relative; path will then be relative to that directory.

        dir_fd may not be implemented on your platform.  If it is unavailable,
        using it will raise a NotImplementedError.

    readv(fd, buffers, /)
        Read from a file descriptor fd into an iterable of buffers.

        The buffers should be mutable buffers accepting bytes.
        readv will transfer data into each buffer until it is full
        and then move on to the next buffer in the sequence to hold
        the rest of the data.

        readv returns the total number of bytes read,
        which may be less than the total capacity of all the buffers.

    register_at_fork(...)
        Register callables to be called when forking a new process.

          before
            A callable to be called in the parent before the fork() syscall.
          after_in_child
            A callable to be called in the child after fork().
          after_in_parent
            A callable to be called in the parent after fork().

        'before' callbacks are called in reverse order.
        'after_in_child' and 'after_in_parent' callbacks are called in order.

    remove(path, *, dir_fd=None)
        Remove a file (same as unlink()).

        If dir_fd is not None, it should be a file descriptor open to a directory,
          and path should be relative; path will then be relative to that directory.
        dir_fd may not be implemented on your platform.
          If it is unavailable, using it will raise a NotImplementedError.

    removedirs(name)
        removedirs(name)
```

```
        Super-rmdir; remove a leaf directory and all empty intermediate
        ones.  Works like rmdir except that, if the leaf directory is
        successfully removed, directories corresponding to rightmost path
        segments will be pruned away until either the whole path is
        consumed or an error occurs.  Errors during this latter phase are
        ignored -- they generally mean that a directory was not empty.

    removexattr(path, attribute, *, follow_symlinks=True)
        Remove extended attribute attribute on path.

        path may be either a string, a path-like object, or an open file descriptor.
        If follow_symlinks is False, and the last element of the path is a symbolic
          link, removexattr will modify the symbolic link itself instead of the file
          the link points to.

    rename(src, dst, *, src_dir_fd=None, dst_dir_fd=None)
        Rename a file or directory.

        If either src_dir_fd or dst_dir_fd is not None, it should be a file
          descriptor open to a directory, and the respective path string (src or dst)
          should be relative; the path will then be relative to that directory.
        src_dir_fd and dst_dir_fd, may not be implemented on your platform.
          If they are unavailable, using them will raise a NotImplementedError.

    renames(old, new)
        renames(old, new)

        Super-rename; create directories as necessary and delete any left
        empty.  Works like rename, except creation of any intermediate
        directories needed to make the new pathname good is attempted
        first.  After the rename, directories corresponding to rightmost
        path segments of the old name will be pruned until either the
        whole path is consumed or a nonempty directory is found.

        Note: this function can fail with the new directory structure made
        if you lack permissions needed to unlink the leaf directory or
        file.

    replace(src, dst, *, src_dir_fd=None, dst_dir_fd=None)
        Rename a file or directory, overwriting the destination.

        If either src_dir_fd or dst_dir_fd is not None, it should be a file
          descriptor open to a directory, and the respective path string (src or dst)
          should be relative; the path will then be relative to that directory.
        src_dir_fd and dst_dir_fd, may not be implemented on your platform.
          If they are unavailable, using them will raise a NotImplementedError.

    rmdir(path, *, dir_fd=None)
        Remove a directory.

        If dir_fd is not None, it should be a file descriptor open to a directory,
          and path should be relative; path will then be relative to that directory.
        dir_fd may not be implemented on your platform.
          If it is unavailable, using it will raise a NotImplementedError.

    scandir(path=None)
        Return an iterator of DirEntry objects for given path.

        path can be specified as either str, bytes, or a path-like object.  If path
        is bytes, the names of yielded DirEntry objects will also be bytes; in
        all other circumstances they will be str.

        If path is None, uses the path='.'.

    sched_get_priority_max(policy)
        Get the maximum scheduling priority for policy.

    sched_get_priority_min(policy)
        Get the minimum scheduling priority for policy.

    sched_getaffinity(pid, /)
        Return the affinity of the process identified by pid (or the current process
if zero).

        The affinity is returned as a set of CPU identifiers.

    sched_getparam(pid, /)
        Returns scheduling parameters for the process identified by pid.

        If pid is 0, returns parameters for the calling process.
        Return value is an instance of sched_param.

    sched_getscheduler(pid, /)
        Get the scheduling policy for the process identified by pid.

        Passing 0 for pid returns the scheduling policy for the calling process.
```

```
sched_rr_get_interval(pid, /)
    Return the round-robin quantum for the process identified by pid, in seconds.

    Value returned is a float.

sched_setaffinity(pid, mask, /)
    Set the CPU affinity of the process identified by pid to mask.

    mask should be an iterable of integers identifying CPUs.

sched_setparam(pid, param, /)
    Set scheduling parameters for the process identified by pid.

    If pid is 0, sets parameters for the calling process.
    param should be an instance of sched_param.

sched_setscheduler(pid, policy, param, /)
    Set the scheduling policy for the process identified by pid.

    If pid is 0, the calling process is changed.
    param is an instance of sched_param.

sched_yield()
    Voluntarily relinquish the CPU.

sendfile(out_fd, in_fd, offset, count)
    Copy count bytes from file descriptor in_fd to file descriptor out_fd.

set_blocking(fd, blocking, /)
    Set the blocking mode of the specified file descriptor.

    Set the O_NONBLOCK flag if blocking is False,
    clear the O_NONBLOCK flag otherwise.

set_inheritable(fd, inheritable, /)
    Set the inheritable flag of the specified file descriptor.

setegid(egid, /)
    Set the current process's effective group id.

seteuid(euid, /)
    Set the current process's effective user id.

setgid(gid, /)
    Set the current process's group id.

setgroups(groups, /)
    Set the groups of the current process to list.

setpgid(pid, pgrp, /)
    Call the system call setpgid(pid, pgrp).

setpgrp()
    Make the current process the leader of its process group.

setpriority(which, who, priority)
    Set program scheduling priority.

setregid(rgid, egid, /)
    Set the current process's real and effective group ids.

setresgid(rgid, egid, sgid, /)
    Set the current process's real, effective, and saved group ids.

setresuid(ruid, euid, suid, /)
    Set the current process's real, effective, and saved user ids.

setreuid(ruid, euid, /)
    Set the current process's real and effective user ids.

setsid()
    Call the system call setsid().

setuid(uid, /)
    Set the current process's user id.

setxattr(path, attribute, value, flags=0, *, follow_symlinks=True)
    Set extended attribute attribute on path to value.

    path may be either a string, a path-like object,  or an open file descriptor.
    If follow_symlinks is False, and the last element of the path is a symbolic
      link, setxattr will modify the symbolic link itself instead of the file
      the link points to.

spawnl(mode, file, *args)
    spawnl(mode, file, *args) -> integer
```

```
    Execute file with arguments from args in a subprocess.
    If mode == P_NOWAIT return the pid of the process.
    If mode == P_WAIT return the process's exit code if it exits normally;
    otherwise return -SIG, where SIG is the signal that killed it.

spawnle(mode, file, *args)
    spawnle(mode, file, *args, env) -> integer

    Execute file with arguments from args in a subprocess with the
    supplied environment.
    If mode == P_NOWAIT return the pid of the process.
    If mode == P_WAIT return the process's exit code if it exits normally;
    otherwise return -SIG, where SIG is the signal that killed it.

spawnlp(mode, file, *args)
    spawnlp(mode, file, *args) -> integer

    Execute file (which is looked for along $PATH) with arguments from
    args in a subprocess with the supplied environment.
    If mode == P_NOWAIT return the pid of the process.
    If mode == P_WAIT return the process's exit code if it exits normally;
    otherwise return -SIG, where SIG is the signal that killed it.

spawnlpe(mode, file, *args)
    spawnlpe(mode, file, *args, env) -> integer

    Execute file (which is looked for along $PATH) with arguments from
    args in a subprocess with the supplied environment.
    If mode == P_NOWAIT return the pid of the process.
    If mode == P_WAIT return the process's exit code if it exits normally;
    otherwise return -SIG, where SIG is the signal that killed it.

spawnv(mode, file, args)
    spawnv(mode, file, args) -> integer

    Execute file with arguments from args in a subprocess.
    If mode == P_NOWAIT return the pid of the process.
    If mode == P_WAIT return the process's exit code if it exits normally;
    otherwise return -SIG, where SIG is the signal that killed it.

spawnve(mode, file, args, env)
    spawnve(mode, file, args, env) -> integer

    Execute file with arguments from args in a subprocess with the
    specified environment.
    If mode == P_NOWAIT return the pid of the process.
    If mode == P_WAIT return the process's exit code if it exits normally;
    otherwise return -SIG, where SIG is the signal that killed it.

spawnvp(mode, file, args)
    spawnvp(mode, file, args) -> integer

    Execute file (which is looked for along $PATH) with arguments from
    args in a subprocess.
    If mode == P_NOWAIT return the pid of the process.
    If mode == P_WAIT return the process's exit code if it exits normally;
    otherwise return -SIG, where SIG is the signal that killed it.

spawnvpe(mode, file, args, env)
    spawnvpe(mode, file, args, env) -> integer

    Execute file (which is looked for along $PATH) with arguments from
    args in a subprocess with the supplied environment.
    If mode == P_NOWAIT return the pid of the process.
    If mode == P_WAIT return the process's exit code if it exits normally;
    otherwise return -SIG, where SIG is the signal that killed it.

stat(path, *, dir_fd=None, follow_symlinks=True)
    Perform a stat system call on the given path.

      path
        Path to be examined; can be string, bytes, a path-like object or
        open-file-descriptor int.
      dir_fd
        If not None, it should be a file descriptor open to a directory,
        and path should be a relative string; path will then be relative to
        that directory.
      follow_symlinks
        If False, and the last element of the path is a symbolic link,
        stat will examine the symbolic link itself instead of the file
        the link points to.

    dir_fd and follow_symlinks may not be implemented
      on your platform.  If they are unavailable, using them will raise a
      NotImplementedError.
```

```
    It's an error to use dir_fd or follow_symlinks when specifying path as
      an open file descriptor.

statvfs(path)
    Perform a statvfs system call on the given path.

    path may always be specified as a string.
    On some platforms, path may also be specified as an open file descriptor.
      If this functionality is unavailable, using it raises an exception.

strerror(code, /)
    Translate an error code to a message string.

symlink(src, dst, target_is_directory=False, *, dir_fd=None)
    Create a symbolic link pointing to src named dst.

    target_is_directory is required on Windows if the target is to be
      interpreted as a directory.  (On Windows, symlink requires
      Windows 6.0 or greater, and raises a NotImplementedError otherwise.)
      target_is_directory is ignored on non-Windows platforms.

    If dir_fd is not None, it should be a file descriptor open to a directory,
      and path should be relative; path will then be relative to that directory.
    dir_fd may not be implemented on your platform.
      If it is unavailable, using it will raise a NotImplementedError.

sync()
    Force write of everything to disk.

sysconf(name, /)
    Return an integer-valued system configuration variable.

system(command)
    Execute the command in a subshell.

tcgetpgrp(fd, /)
    Return the process group associated with the terminal specified by fd.

tcsetpgrp(fd, pgid, /)
    Set the process group associated with the terminal specified by fd.

times()
    Return a collection containing process timing information.

    The object returned behaves like a named tuple with these fields:
      (utime, stime, cutime, cstime, elapsed_time)
    All fields are floating point numbers.

truncate(path, length)
    Truncate a file, specified by path, to a specific length.

    On some platforms, path may also be specified as an open file descriptor.
      If this functionality is unavailable, using it raises an exception.

ttyname(fd, /)
    Return the name of the terminal device connected to 'fd'.

    fd
      Integer file descriptor handle.

umask(mask, /)
    Set the current numeric umask and return the previous umask.

uname()
    Return an object identifying the current operating system.

    The object behaves like a named tuple with the following fields:
      (sysname, nodename, release, version, machine)

unlink(path, *, dir_fd=None)
    Remove a file (same as remove()).

    If dir_fd is not None, it should be a file descriptor open to a directory,
      and path should be relative; path will then be relative to that directory.
    dir_fd may not be implemented on your platform.
      If it is unavailable, using it will raise a NotImplementedError.

unsetenv(name, /)
    Delete an environment variable.

urandom(size, /)
    Return a bytes object containing random bytes suitable for cryptographic use.

utime(...)
    Set the access and modified time of path.

    path may always be specified as a string.
```

```
    On some platforms, path may also be specified as an open file descriptor.
      If this functionality is unavailable, using it raises an exception.

    If times is not None, it must be a tuple (atime, mtime);
        atime and mtime should be expressed as float seconds since the epoch.
    If ns is specified, it must be a tuple (atime_ns, mtime_ns);
        atime_ns and mtime_ns should be expressed as integer nanoseconds
        since the epoch.
    If times is None and ns is unspecified, utime uses the current time.
    Specifying tuples for both times and ns is an error.

    If dir_fd is not None, it should be a file descriptor open to a directory,
      and path should be relative; path will then be relative to that directory.
    If follow_symlinks is False, and the last element of the path is a symbolic
      link, utime will modify the symbolic link itself instead of the file the
      link points to.
    It is an error to use dir_fd or follow_symlinks when specifying path
      as an open file descriptor.
    dir_fd and follow_symlinks may not be available on your platform.
      If they are unavailable, using them will raise a NotImplementedError.

wait()
    Wait for completion of a child process.

    Returns a tuple of information about the child process:
        (pid, status)

wait3(options)
    Wait for completion of a child process.

    Returns a tuple of information about the child process:
      (pid, status, rusage)

wait4(pid, options)
    Wait for completion of a specific child process.

    Returns a tuple of information about the child process:
      (pid, status, rusage)

waitid(idtype, id, options, /)
    Returns the result of waiting for a process or processes.

      idtype
        Must be one of be P_PID, P_PGID or P_ALL.
      id
        The id to wait on.
      options
        Constructed from the ORing of one or more of WEXITED, WSTOPPED
        or WCONTINUED and additionally may be ORed with WNOHANG or WNOWAIT.

    Returns either waitid_result or None if WNOHANG is specified and there are
    no children in a waitable state.

waitpid(pid, options, /)
    Wait for completion of a given child process.

    Returns a tuple of information regarding the child process:
        (pid, status)

    The options argument is ignored on Windows.

waitstatus_to_exitcode(status)
    Convert a wait status to an exit code.

    On Unix:

    * If WIFEXITED(status) is true, return WEXITSTATUS(status).
    * If WIFSIGNALED(status) is true, return -WTERMSIG(status).
    * Otherwise, raise a ValueError.

    On Windows, return status shifted right by 8 bits.

    On Unix, if the process is being traced or if waitpid() was called with
    WUNTRACED option, the caller must first check if WIFSTOPPED(status) is true.
    This function must not be called if WIFSTOPPED(status) is true.

walk(top, topdown=True, onerror=None, followlinks=False)
    Directory tree generator.

    For each directory in the directory tree rooted at top (including top
    itself, but excluding '.' and '..'), yields a 3-tuple

        dirpath, dirnames, filenames

    dirpath is a string, the path to the directory.  dirnames is a list of
    the names of the subdirectories in dirpath (excluding '.' and '..').
    filenames is a list of the names of the non-directory files in dirpath.
```

```
        Note that the names in the lists are just names, with no path components.
        To get a full path (which begins with top) to a file or directory in
        dirpath, do os.path.join(dirpath, name).

        If optional arg 'topdown' is true or not specified, the triple for a
        directory is generated before the triples for any of its subdirectories
        (directories are generated top down).  If topdown is false, the triple
        for a directory is generated after the triples for all of its
        subdirectories (directories are generated bottom up).

        When topdown is true, the caller can modify the dirnames list in-place
        (e.g., via del or slice assignment), and walk will only recurse into the
        subdirectories whose names remain in dirnames; this can be used to prune the
        search, or to impose a specific order of visiting.  Modifying dirnames when
        topdown is false has no effect on the behavior of os.walk(), since the
        directories in dirnames have already been generated by the time dirnames
        itself is generated. No matter the value of topdown, the list of
        subdirectories is retrieved before the tuples for the directory and its
        subdirectories are generated.

        By default errors from the os.scandir() call are ignored.  If
        optional arg 'onerror' is specified, it should be a function; it
        will be called with one argument, an OSError instance.  It can
        report the error to continue with the walk, or raise the exception
        to abort the walk.  Note that the filename is available as the
        filename attribute of the exception object.

        By default, os.walk does not follow symbolic links to subdirectories on
        systems that support them.  In order to get this functionality, set the
        optional argument 'followlinks' to true.

        Caution:  if you pass a relative pathname for top, don't change the
        current working directory between resumptions of walk.  walk never
        changes the current directory, and assumes that the client doesn't
        either.

        Example:

        import os
        from os.path import join, getsize
        for root, dirs, files in os.walk('python/Lib/email'):
            print(root, "consumes", end="")
            print(sum(getsize(join(root, name)) for name in files), end="")
            print("bytes in", len(files), "non-directory files")
            if 'CVS' in dirs:
                dirs.remove('CVS')  # don't visit CVS directories

    write(fd, data, /)
        Write a bytes object to a file descriptor.

    writev(fd, buffers, /)
        Iterate over buffers, and write the contents of each to a file descriptor.

        Returns the total number of bytes written.
        buffers must be a sequence of bytes-like objects.

DATA
    CLD_CONTINUED = 6
    CLD_DUMPED = 3
    CLD_EXITED = 1
    CLD_KILLED = 2
    CLD_STOPPED = 5
    CLD_TRAPPED = 4
    EX_CANTCREAT = 73
    EX_CONFIG = 78
    EX_DATAERR = 65
    EX_IOERR = 74
    EX_NOHOST = 68
    EX_NOINPUT = 66
    EX_NOPERM = 77
    EX_NOUSER = 67
    EX_OK = 0
    EX_OSERR = 71
    EX_OSFILE = 72
    EX_PROTOCOL = 76
    EX_SOFTWARE = 70
    EX_TEMPFAIL = 75
    EX_UNAVAILABLE = 69
    EX_USAGE = 64
    F_LOCK = 1
    F_OK = 0
    F_TEST = 3
    F_TLOCK = 2
    F_ULOCK = 0
    GRND_NONBLOCK = 1
    GRND_RANDOM = 2
    MFD_ALLOW_SEALING = 2
```

```
MFD_CLOEXEC = 1
MFD_HUGETLB = 4
MFD_HUGE_16GB = -2013265920
MFD_HUGE_16MB = 1610612736
MFD_HUGE_1GB = 2013265920
MFD_HUGE_1MB = 1342177280
MFD_HUGE_256MB = 1879048192
MFD_HUGE_2GB = 2080374784
MFD_HUGE_2MB = 1409286144
MFD_HUGE_32MB = 1677721600
MFD_HUGE_512KB = 1275068416
MFD_HUGE_512MB = 1946157056
MFD_HUGE_64KB = 1073741824
MFD_HUGE_8MB = 1543503872
MFD_HUGE_MASK = 63
MFD_HUGE_SHIFT = 26
NGROUPS_MAX = 65536
O_ACCMODE = 3
O_APPEND = 1024
O_ASYNC = 8192
O_CLOEXEC = 524288
O_CREAT = 64
O_DIRECT = 16384
O_DIRECTORY = 65536
O_DSYNC = 4096
O_EXCL = 128
O_LARGEFILE = 0
O_NDELAY = 2048
O_NOATIME = 262144
O_NOCTTY = 256
O_NOFOLLOW = 131072
O_NONBLOCK = 2048
O_PATH = 2097152
O_RDONLY = 0
O_RDWR = 2
O_RSYNC = 1052672
O_SYNC = 1052672
O_TMPFILE = 4259840
O_TRUNC = 512
O_WRONLY = 1
POSIX_FADV_DONTNEED = 4
POSIX_FADV_NOREUSE = 5
POSIX_FADV_NORMAL = 0
POSIX_FADV_RANDOM = 1
POSIX_FADV_SEQUENTIAL = 2
POSIX_FADV_WILLNEED = 3
POSIX_SPAWN_CLOSE = 1
POSIX_SPAWN_DUP2 = 2
POSIX_SPAWN_OPEN = 0
PRIO_PGRP = 1
PRIO_PROCESS = 0
PRIO_USER = 2
P_ALL = 0
P_NOWAIT = 1
P_NOWAITO = 1
P_PGID = 2
P_PID = 1
P_PIDFD = 3
P_WAIT = 0
RTLD_DEEPBIND = 8
RTLD_GLOBAL = 256
RTLD_LAZY = 1
RTLD_LOCAL = 0
RTLD_NODELETE = 4096
RTLD_NOLOAD = 4
RTLD_NOW = 2
RWF_DSYNC = 2
RWF_HIPRI = 1
RWF_NOWAIT = 8
RWF_SYNC = 4
R_OK = 4
SCHED_BATCH = 3
SCHED_FIFO = 1
SCHED_IDLE = 5
SCHED_OTHER = 0
SCHED_RESET_ON_FORK = 1073741824
SCHED_RR = 2
SEEK_CUR = 1
SEEK_DATA = 3
SEEK_END = 2
SEEK_HOLE = 4
SEEK_SET = 0
ST_APPEND = 256
ST_MANDLOCK = 64
ST_NOATIME = 1024
ST_NODEV = 4
ST_NODIRATIME = 2048
```

```
    ST_NOEXEC = 8
    ST_NOSUID = 2
    ST_RDONLY = 1
    ST_RELATIME = 4096
    ST_SYNCHRONOUS = 16
    ST_WRITE = 128
    TMP_MAX = 238328
    WCONTINUED = 8
    WEXITED = 4
    WNOHANG = 1
    WNOWAIT = 16777216
    WSTOPPED = 2
    WUNTRACED = 2
    W_OK = 2
    XATTR_CREATE = 1
    XATTR_REPLACE = 2
    XATTR_SIZE_MAX = 65536
    X_OK = 1
    __all__ = ['altsep', 'curdir', 'pardir', 'sep', 'pathsep', 'linesep', ...
    altsep = None
    confstr_names = {'CS_GNU_LIBC_VERSION': 2, 'CS_GNU_LIBPTHREAD_VERSION'...
    curdir = '.'
    defpath = '/bin:/usr/bin'
    devnull = '/dev/null'
    environ = environ({'SHELL': '/bin/bash', 'SESSION_MANAGER'...D': 'modu...
    environb = environ({b'SHELL': b'/bin/bash', b'SESSION_MANAG...': b'mod...
    extsep = '.'
    linesep = '\n'
    name = 'posix'
    pardir = '..'
    pathconf_names = {'PC_ALLOC_SIZE_MIN': 18, 'PC_ASYNC_IO': 10, 'PC_CHOW...
    pathsep = ':'
    sep = '/'
    supports_bytes_environ = True
    sysconf_names = {'SC_2_CHAR_TERM': 95, 'SC_2_C_BIND': 47, 'SC_2_C_DEV'...

FILE
    /usr/lib/python3.9/os.py
```

snipped output:

```
Help on module os:

NAME
    os - OS routines for NT or Posix depending on what system we're on.

MODULE REFERENCE
    https://docs.python.org/3.9/library/os

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.
```

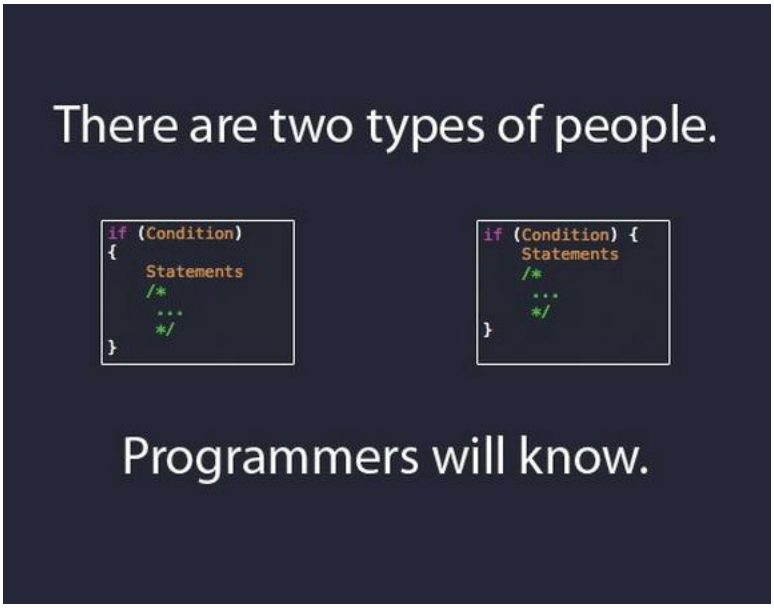
```
# Help utility on getcwd function of sys module
help(os.getcwd)
```

```
Help on built-in function getcwd in module posix:

getcwd()
    Return a unicode string representing the current working directory.
```

🔔 Help function returns the docstrings associated with the respective Modules, Keywords or functions.

13. Indendation

I have seen memes of people fighting about opening braces, whether they should be starting in the same line or in next line in the programming languages like C, Java etc… 👻



Python Developers be like: Hold my Beer 🍺

In Python, we don't use curly braces for grouping the statements. Instead, we use Indendation.

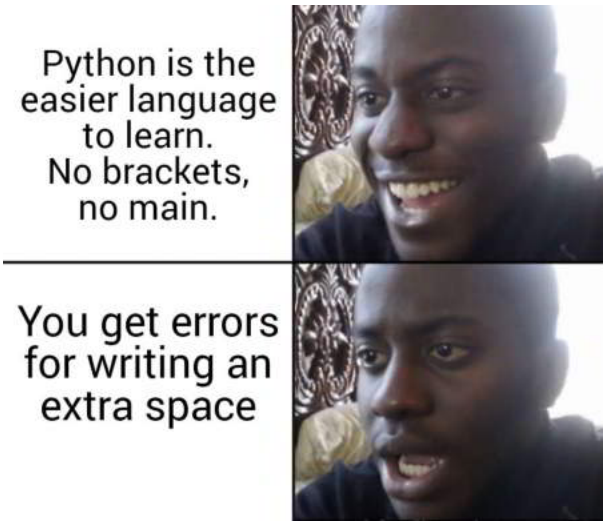
Each group of statements are indended using spaces or tabs.

```
class Example:
    # Every method belonging to a class must be indended equally.
    def __init__(self):
        name = "indendation example"

    def check_for_odd_or_even(self, number: int):
        # Everything that belongs to this method are indended as well.
        if number % 2 == 0:
            print(f"{number} is even.")
        else:
            print(f"{number} is odd.")


# We can see that the say_hello_multiple_times is not indended inside the Example
class.
# Hence, say_hello_multiple_times function doesn't belong to Example class.
def say_hello_multiple_times(count: int):
    for _ in range(count):
        # Loops or conditions are also needed to be intended.
        print("Hello")
```

PEP-8 recommends to use **4 Spaces instead of Tabs**. Although using of Tabs do work, but ensure **not** to mix both tabs and spaces, as you might get `TabError` for such indendations.

13.1. A meme on indendation 😜

If using a normal text editor like notepad where it doesn't show the warnings or errors, sometimes we might get errors due to wrong indendation or mix usage of both tabs and spaces, we get an error and it would be tricky to resolve it as it is invisible.



14. Comments and Docstrings

Comments are used to explain the code. The interpreter doesn't execute the comments. There are 3 types of comments in Python:

- Single Line
- In-Line
- Multi Line

14.1. Single Line

Single line comments are written in a single line. Single line comments start with `#`

```
# Here's a single line comment
```

14.2. In-Line

In-Line comments are written beside the code.

```
print("Hello")  # Here's a In-Line comment
```

14.3. Multi Line

Sometimes we need to write a huge explanation using comments, in those cases we do use multi-line comments. multiline comments are enclosed in `""" """` or `''' '''`

```
"""
Here's a
multiline comment.
"""
```

14.4. Docstrings

Docstrings are specific type of comments that are stored as a attribute to the module, class, method or function.

Docstrings are written similar to the multi-line comments using `""" """` or `''' '''`, the only difference would be they are written exactly at the start(first statement) of the module, class, method or function.

Docstrings can be programatically acccessed using the `__doc__` method or through the built-in function `help`. Let's give a try 😎.

```
def double_the_value(value: int):
    """Doubles the integer value passed to the function and returns it."""
    return value * 2
```

14.4.1. Using help

`help` function provides the docstrings as well as the information about the module, class, method or function.

```
help(double_the_value)
```

```
Help on function double_the_value in module __main__:

double_the_value(value: int)
    Doubles the integer value passed to the function and returns it.
```

14.4.2. Using `__doc__`

```
print(double_the_value.__doc__)
```

```
Doubles the integer value passed to the function and returns it.
```

Can we use the single line comments instead of multi-line docstrings 🤔? Let's try this as well.

```
def test_single_line_comment_as_docstring():
    # This is a single-line comment
    pass
```

```
print(test_single_line_comment_as_docstring.__doc__)
```

```
None
```

We can see that `None` is printed, which explains that we can't use single-line comments as docstrings 🙂

14.5. Docstrings for documentation of code.

PEP-257 defines two types of docstrings.

- One-Line docstring
- Multi-Line docstring

14.5.1. One-Line docstring

One-line docstrings are suited for short and simple Modules, classes, methods or functions.

```
def one_line_docstring():
    """This is a one-line docstring"""
    pass
```

14.5.2. Multi-Line docstring

Multi-line docstrings are suited for long, complex Modules, classes, methods or functions

```python
def multi_line_docstring(arg1: int, arg2: str) -> None:
    """
    This is a multi-line docstring.

    Arguments:
      arg1 (int): Argument 1 is an integer.
      arg2 (str): Argument 2 is a string.
    """
    pass
```

14.6. Styles of docstrings

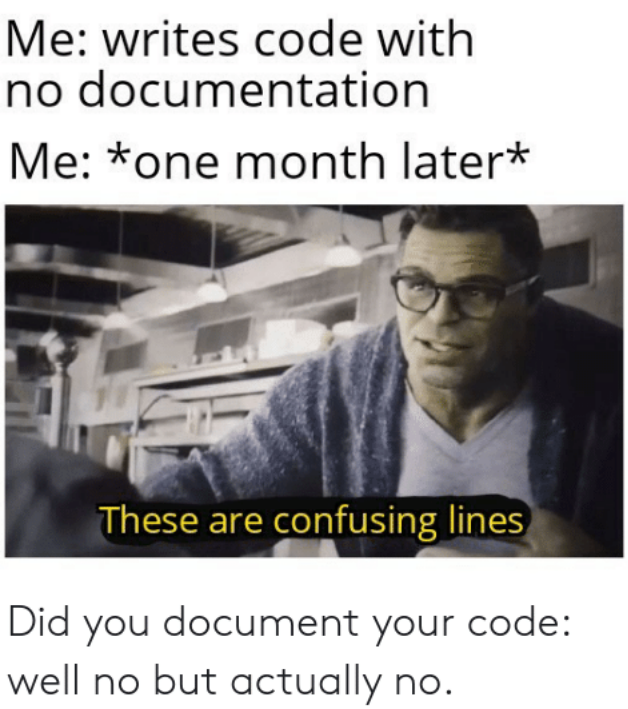
There are multiple styles of writing docstrings such as reStructuredText, Google Python Style Guide, Numpy style.

We could use any of the above docstrings style as long as we stay consistent.

Sphinx is a tool that generated beautiful HTML based documentation 📜 from the docstrings we provide in our code. reStructuredText is the default style, for other styles like Google Python style, numpy we could use plugins like Napoleon.

Sphinx also provides various templates we can choose from to create the HTML documentation out of it. 😎♥

14.6.1. A meme on Documentation 😂



It's always good and professional to have our code documented 🙂.

15. Functions

The purpose of Functions is grouping the code into organised, readable and reusable format. By using the functions, code redundancy can be reduced.

A soft rule of functions is that, an function shoud be **Small** and **Do One Thing** mentioned in Clean Code by Robert C. Martin.

Functions are nothing new to us. We have already used `print` function in our previous lessons, just that it is a built-in function. There are other built-in functions like `help`, `len`, `sorted`, `map`, `filter`, `reduce` etc…

In Python functions are created by using the keyword `def` followed by the function name and if required parameters.

```
def function_name(parameters):
    # statements...
    ...
    ...
    ...
```

Here `function_name` is the identifier for the function through which it can be called. `parameters` are optional in the function signature. A function may have any number of parameters to be bound to the function. As we already know we do use Indendation to group the statments, all the statements belonging to the function are indended in the function.

By convention function names should be in camelcase 🐫 and be a verb.

Let's get started with a basic function

15.1. Simple function

```
def greet():
    print("Hello Pythoneer! 😎")
```

```
greet()  # Calling the function
```

```
Hello Pythoneer! 😎
```

This is a pretty basic function which just prints to the console saying "Hello Pythoneer!😎", as we are not returning anything using the `return` keyword, our function `greet` implicitly returns `None` object.

We could return objects from the function using the keyword `return`

```
def greet():
    return "Hello Pythoneer! 😎"
```

```
greet_word = greet()
print(greet_word)
```

```
Hello Pythoneer! 😎
```

15.2. Functions - First Class Objecs

In Python 🐍, Functions are First class objects. There are a few criterias defined for an object to be First class object like functions can be passed as argument, assigned to a variable, return a function.

15.2.1. Passing the function to a different function

```
def first_function():
    print("One Pint of 🍺")
```

```
def second_function(func):
    func()
    print("Two Pints of beer 🍺🍺")
```

```
second_function(first_function)
```

```
One Pint of 🍺
Two Pints of beer 🍺🍺
```

Yippeee! We have successfully passed our `first_function` to the `second_function` where `first_function` is being called inside the `second_function`

15.2.2. Assigning the function to a variable

```
# note that we are not calling the function, we are just assigning,
# if we call the function, the returned value would be assigned to our variable.
i_am_a_variable = first_function
```

```
i_am_a_variable()
```

```
One Pint of 🍺
```

15.2.3. Returning a function

```
def lets_return_a_function():
    return first_function
```

```
obj = lets_return_a_function()
print(f"obj is {obj}")
print(f"obj name is {obj.__name__}")
print(f"Is obj callable? {callable(obj)}")
```

```
obj is <function first_function at 0x7fc374462310>
obj name is first_function
Is obj callable? True
```

Cheers again 🍻! We accomplished mission of returning the function. In the above example, we are printing the the `obj` itself which provides the `__str__` representation of the obj, next we are printing the name of the `obj` which is the function name itself, and finally we are checking if our `obj` is callable, if an object is callable, then `callable` function returns `True` else `False`

15.2.4. Deletion of function object

As we already know that everything in Python is an object, even function as well is an object. We can even delete our function using the `del` keyword.

```
del first_function
first_function()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_5029/186491708.py in <module>
      1 del first_function
----> 2 first_function()

NameError: name 'first_function' is not defined
```

As we deleted the `first_function`, if we try to call that function, we do get to see `NameError` saying `first_function` is not defined.

15.3. Types of Arguments

As we already know we can pass the parameters to the function, we can give a try on those too.. But before trying out, let's know about the types of Arguments we can define in the function signature. We have the below 4 types of Arguments:

- Positional Arguments
- Unnamed positional Arguments / VarArgs
- Keyword-only Arguments
- Keyword arguments / Varkwargs

16. Positional Arguments

```
def add(operand_1, operand_2):
    print(f"The sum of {operand_1} and {operand_2} is {operand_1 + operand_2}")
```

Yipeee! we have created a new function called add which is expected to add two integers values, Just kidding 😜, thanks to the dynamic typing of the Python, we can even add float values, concat strings and many more using our `add` function, but for now, let's stick with the addition of integers 😎

```
add(1, 3)
```

```
The sum of 1 and 3 is 4
```

Yup, we did got our result ⭐. what if I forget passing a value? we would see a `TypeError` exception raised 👻

```
add(1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_5046/746715703.py in <module>
----> 1 add(1)

TypeError: add() missing 1 required positional argument: 'operand_2'
```

The name **Positional arguments** itself says the arguments should be according to the function signature. But here's a deal, we can change the order of arguments being passed, just that we should pass them with the respective keyword 🙂

`Example`

```
def difference(a, b):
    print(f"The difference of {b} from {a} is {a - b}")
```

```
difference(5, 8)
```

```
The difference of 8 from 5 is -3
```

```
difference(b=8, a=5)  # Positions are swapped, but passing the objects as keywords.
```

```
The difference of 8 from 5 is -3
```

We can see in the above example that even if the positions are changed, but as we have are passing them through keywords, the result remains the same. ⭐

16.1. Position only arguments

We do have the power ✊ to make the user call the function's position only arguments the way we want, Thanks to PEP-570 for Python >= 3.8

The syntax defined by the PEP-570 regarding Position only arguments is as:

```
def name(positional_only_parameters, /, positional_or_keyword_parameters, *,
keyword_only_parameters):
```

```
def greet(greet_word, /, name_of_the_user):
    print(f"{greet_word} {name_of_the_user}!")
```

In the above example, we do have two arguments `greet_word` and `name_of_the_user` we used `/` to say that **Hey Python! Consider `greet_word` as Positional only Argument**

When we try to call our function `greet` with greet_word as keyword name, Boom 💣, we get a `TypeError` exception.

```
greet(greet_word="Hello", name_of_the_user="Pythonist")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_5046/1994685720.py in <module>
----> 1 greet(greet_word="Hello", name_of_the_user="Pythonist")

TypeError: greet() got some positional-only arguments passed as keyword arguments:
'greet_word'
```

Try to call our `greet` with `greet_word` as positional only argument, meaning not passing it by keyword name. We can hope that there won't be any exception raised. 😁

```
# Calling greet function with name_of_the_user as Positional keyword.
greet("Hello", "Pythonist")

# Calling greet function with name_of_the_user with keyword name.
greet("Hello", name_of_the_user="Pythoneer😍")
```

```
Hello Pythonist!
Hello Pythoneer😍!
```

17. Unnamed Positional Arguments

let's think we need to count the number of bucks I spent from the past 3 days. I could write a function as below:

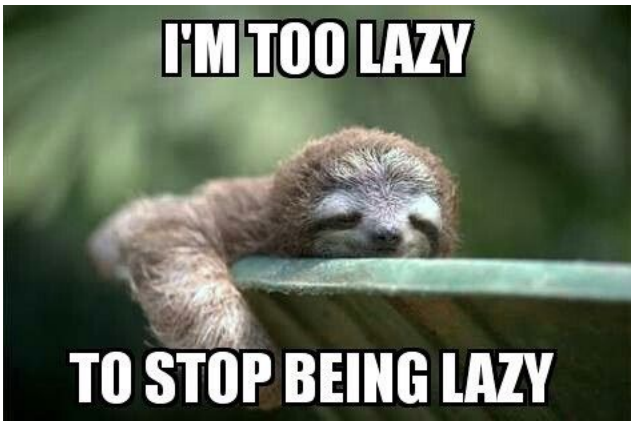
```
def count_my_expenses_for_last_3_days(day_1, day_2, day_3):
    print(f"The total expenses for last 3 days is : {day_1 + day_2 + day_3}")
```

```
count_my_expenses_for_last_3_days(88, 12, 15)
```

```
The total expenses for last 3 days is : 115
```

The day passed down and now I want to find my expenses of my last 4 days, but I am too lazy to write new function like:

```
def count_my_expenses_for_last_4_days(day_1, day_2, day_3, day_4):
    print(f"The total expenses for last 4 days is : {day_1 + day_2 + day_3 + day_4}")
```



And I much more lazier to modify the function each day. No worries, we have Unnamed Positional arguments as our Saviour in this case.

Sometimes we might not know the number of arguments we need to send to a function. Using Unnamed Positional Arguments we can pass any number of arguments to the function. The function receives all the arguments placed in the tuple.

17.1. Using Unnamed Positional Arguments to find our expenses

```python
def count_my_expenses(*expenses):
    # We could use sum function like sum(expenses). but for now let's go the raw way.
    total = 0
    for expense in expenses:
        total += expense
    print(f"Total expenses for last {len(expenses)} is {total}")
```

For 3 days:

```python
count_my_expenses(100, 23, 4544)
```

```
Total expenses for last 3 is 4667
```

For 5 days:

```python
count_my_expenses(100, 23, 4544, 4, 13)
```

```
Total expenses for last 5 is 4684
```

For 8 days:

```python
count_my_expenses(100, 23, 4544, 4, 13, 34, 86, 123)
```

```
Total expenses for last 8 is 4927
```

Hence we can see that for any number of days of expenses our function `count_my_expenses` works great 🤖 🍾.

We can even pass the already present objects in a iterable to our function, just that we need to unpack the iterable using the `*`

```python
my_expenses = [100, 23, 4544, 4, 13, 34, 86, 123]
count_my_expenses(*my_expenses)
```

```
Total expenses for last 8 is 4927
```

let's check what is the datatype of the the Unnamed positional arguments passed tp the function

```python
def example(*args):
    print(f"The datatype of args is {type(args)}")
    print(f"The contents of the args are: {args}")


# Calling the function.
example("abc")
```

```
The datatype of args is <class 'tuple'>
The contents of the args are: ('abc',)
```

Yup! The datatype of Unnamed Positional arguments is `Tuple`, and the objects passed as args are placed in the tuple object. 🙂

🔔 By the way, this is not our first time using Unnamed Positional arguments. We have already used `print` function many times and it accepts Unnamed Positional arguments to be printed.

```python
print("Hello", "Pythonist!", "⭐")
```

```
Hello Pythonist! ⭐
```

18. Keyword-only arguments

Few times being explicit is better which increases the readability of code. If a function signature has Keyword-only arguments, then while caling the function, we need to pass our objects by their keyword names. PEP-3102 defines the Keyword-only arguments.

Well, how to define the keyword only arguments 🤔? In the previous lesson about Positonal Arguments we have seen that Positional-only Arguments whose function signature is created by using `/`. Similarly for Keyword-only Argument, we use `*` in the signature.

```
def keyword_only_argument_signature(*, arg1, arg2):
    ...
```

`Example:`

```
def greet(*, greet_word, name):
    print(f"{greet_word} {name}!")
```

Now if we want to try calling our new function `greet` as `greet("Hello", "Pythonist♥")`, we should be seeing a `TypeError`.

```
greet("Hello", "Pythonist ♥")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_5114/1295827636.py in <module>
----> 1 greet("Hello", "Pythonist ♥")

TypeError: greet() takes 0 positional arguments but 2 were given
```

The only way we can call our `greet` function is by passing our both `greet_word` and `name` values with keyword names.

```
greet(greet_word="Hello", name="Pythonist ♥")
```

```
Hello Pythonist ♥!
```

19. Keyword arguments

The Keyword Arguments name suggests that they are called through names when calling the function.

As we saw Unnamed Positional arguments already, Keyword arguments are similar, they can be passed any number of objects, the only difference would be they needed to be passed with keyword names. To define the keyword arguments in a function signature, we need to prefix `**` for the argument.

```
def example_keyword_arguments(**kwargs):
    print(kwargs)
```

```
example_keyword_arguments(key1="value1", key2="value2")
```

```
{'key1': 'value1', 'key2': 'value2'}
```

We can even pass a dictionary as well 😎, just that we need to pass the dictionary with unpacking them as `**`

```
my_dictionary = {"key1": "value1", "key2": "value2"}
example_keyword_arguments(**my_dictionary)
```

```
{'key1': 'value1', 'key2': 'value2'}
```

If we try to pass objects as positional parameters, we would be seeing our friend `TypeError` being raised 👻

```
example_keyword_arguments("Hello")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_5131/291417681.py in <module>
----> 1 example_keyword_arguments("Hello")

TypeError: example_keyword_arguments() takes 0 positional arguments but 1 was given
```

20. Default Arguments

Default arguments are the ones when calling the function, no object is given in its place, its default value would be considered 🔔.

Default arguments are assigned in the function signature as

```
def func(arg=<obj>):
```

Let's get through an example

```
def greet(greet_word="Hello", name="Pythonist!"):
    print(f"{greet_word} {name}!")
```

```
greet("Hey")
```

```
Hey Pythonist!!
```

We see that the output for the above example is `Hey Pythonist!`, we have passed just `Hey` for `greet_word` argument, as we have passed object for `greet_word`, it took "Hey" as its value, but coming to `name` argument, we haven't passed any value to it, so it took default object for `name` as `Pythonist!`

20.1. Note on default arguments with respect to mutable objects

In the previous example, we have made default argument values to be string objects, and we already know that string objects are immutable objects and it works really fine as we expected.

But what would be result if we use mutable objects like lists, dictionary 🤔?

```
def test_mutable_objects_as_default_argument(my_list=[]):
    my_list.append("🍰")
    print(my_list)
```

```
test_mutable_objects_as_default_argument()  # Calling the function for the first time.
test_mutable_objects_as_default_argument()  # Calling the function for the second time.
```

```
['🍰']
['🍰', '🍰']
```

Ouch, as we are not passing any argument during calling our `test_mutable_objects_as_default_argument` function, both the times we expected the result should be the same. But, we do see that during second time calling of the function, there is one extra 🍰 present in the output.

I would be happy for getting an extra cake in my plate 😋, but not in the above output. Well, the problem is that, `my_list` in the `test_mutable_objects_as_default_argument` is being stored as the function attribute and being persisted and mutated everytime function is called.

We could see the default values of our function using the `__defaults__` method.

```
test_mutable_objects_as_default_argument.__defaults__
```

```
(['🍰', '🍰'],)
```

We see there are cake objects being stored in the defaults of the function, No worries, we do have a fix for that.

`Solution:` Use `None` object as default argument.

```
def test_again_as_default_argument_using_none(my_list=None):
    if my_list is None:
        my_list = []
    my_list.append("🍰")
    print(my_list)
```

```
test_again_as_default_argument_using_none()  # Calling the function for the first time.
test_again_as_default_argument_using_none()  # Calling the function for the second
time.
```

```
['🍰']
['🍰']
```

Hurray 🍾! We learned how to deal with default arguments for mutable objects.

If we pass a list containing an objects, our cake 🍰 would be appended and printed.

```
test_again_as_default_argument_using_none(["🍷"])
```

```
['🍷', '🍰']
```

20.2. Tidbits 💡

We can't assign default arguments to Unnamed positional arguments (VarArgs) and Keyword arguments as there are optional in first place with default values as empty tuple `()` for Unnamed positional arguments and empty dictionary `{}` for Keyword arguments. If we try assigning default argument to Unnamed positional arguments r Keyword arguments, we would see `SyntaxError` spawned 👻.

21. TLDR regarding function arguments 💡

Till now we have seen all the 4 types of arguments that we can use in functions.

- Positional Arguments
- Unnamed positional Arguments / VarArgs
- Keyword-only Arguments
- Keyword arguments / Varkwargs

Let's give it a shot with all the above arguments in a function 😎

The complete syntax of a function eliding varargs and keyword arguments defined in PEP-570 would be as:

```
def name(positional_only_parameters, /, positional_or_keyword_parameters, *,
keyword_only_parameters):
```

`Example:`

Let's build a complete function with all the types of arguments 😎.

```
def function(positional_only, /, position="🐍", *varargs, keyword_only, **keyword):
    print(f"{positional_only=}")
    print(f"{position=}")
    print(f"{varargs=}")
    print(f"{keyword_only=}")
    print(f"{keyword=}")

    # datatype of varargs and keyword.
    print(f"The datatype of varargs is {type(varargs)}")
    print(f"The datatype of keyword is {type(keyword)}")
```

Let's call our beautiful function ♥

```
function(
    "Python",
    "♥",
    "Python",
    "is",
    "Cool",
    keyword_only="😊",
    key1="value1",
    key2="value2",
)
```

```
positional_only='Python'
position='♥'
varargs=('Python', 'is', 'Cool')
keyword_only='😋'
keyword={'key1': 'value1', 'key2': 'value2'}
The datatype of varargs is <class 'tuple'>
The datatype of keyword is <class 'dict'>
```

The above calling of function can also be written as:

```
function(
    "Python",
    "♥",
    *["Python", "is", "Cool"],
    keyword_only="😋",
    **{"key1": "value1", "key2": "value2"},
)  # Unpacking.
```

```
positional_only='Python'
position='♥'
varargs=('Python', 'is', 'Cool')
keyword_only='😋'
keyword={'key1': 'value1', 'key2': 'value2'}
The datatype of varargs is <class 'tuple'>
The datatype of keyword is <class 'dict'>
```

Let's make `position` be it's default value.

```
function(
    "Python",
    *["Python", "is", "Cool"],
    keyword_only="😋",
    **{"key1": "value1", "key2": "value2"},
)
```

```
positional_only='Python'
position='Python'
varargs=('is', 'Cool')
keyword_only='😋'
keyword={'key1': 'value1', 'key2': 'value2'}
The datatype of varargs is <class 'tuple'>
The datatype of keyword is <class 'dict'>
```

Ouch, we see that `position` has taken the `Python` as it's value which we intended to be one of the value of our `varargs`. The solution for this is to pass our `*["Python", "is", "Cool"]` as keyword argument like `varargs=["Python", "is", "Cool"]`. **NOTE** that there won't be unpacking symbol `*` here.

```
function(
    "Python",
    varargs=["Python", "is", "Cool"],
    keyword_only="😋",
    **{"key1": "value1", "key2": "value2"},
)
```

```
positional_only='Python'
position='🐍'
varargs=()
keyword_only='😋'
keyword={'varargs': ['Python', 'is', 'Cool'], 'key1': 'value1', 'key2': 'value2'}
The datatype of varargs is <class 'tuple'>
The datatype of keyword is <class 'dict'>
```

We can even notice that in the above example, we have passed `varargs=["Python", "is", "Cool"]`, but in the output the datatype of varargs is printed as tuple. Not just in above example, in all the above examples, we can see that `varargs` is `tuple` and `keyword` is `dictionary`.

💡 Unnamed Positional arguments datatype is always tuple and keyword argument datatype is always dictionary .

22. Lambda Functions

Lambda functions are inline functions which have only 1 statement. They are created by using the keyword `lambda` They do not have any name, so they are also known as Anonymous functions. Although they don't have a name, they can be bound to a variable.

A simple lambda function to greets us 👋:

```
greet = lambda: "Hello Pythonist!"
```

```
print(greet())
```

```
Hello Pythonist!
```

The above lambda function can be rewritten as a regular function as:

```
def greet():
    return "Hello Pythonist!"
```

```
print(greet())
```

```
Hello Pythonist!
```

Conceptually, lambda functions are similar to regular functions which are defined using `def`, just that lambda function accepts only 1 statement.

Let's try calling lambda function without assigning the function to any variable.

```
print((lambda: "Hello Pythonist! ♥")())
```

```
Hello Pythonist! ♥
```

In the above example, we are creating the lambda expression enclosed in paranthesis and calling the function by using `()` at the end. As there is no name for the lambda function we just called, this is the reason why Lambda expressions/functions are also called as Anonymous functions.

We can pass parameters as well to the lambda function

```
add = lambda a, b: a + b
print(add(3, 5))
```

```
8
```

Till now, everything about `lambda functions` and `regular functions` do look the same, Is there any differnce? Yup, here it is, Lambda functions do have the Lexical closures similar to loops in regular functions. What the heck is Lexical closure 🤔? At the end of the lexical scope, the value is stil remembered unlike in the programming languages C, Golang etc.. Let's try it out with an example 🙂

```
def do_sum(value):
    return lambda a: a + value
```

```
adder_3 = do_sum(3)
adder_10 = do_sum(10)
```

```
print(adder_3(5))
print(adder_10(5))
```

```
8
15
```

Here we can see that `adder_3` and `adder_10` are persisting the values that `3` and `10` that we passed during calling the `do_sum` function which returned the lambda function which holds our 3 and 10.

By Prodigious Python