

Functions

| | | |
|---|--|---|
| <code>def my_function(first: int, second: str) -> float:</code> | | |
| <code>sum: Callable[[int, int], int] = lambda x, y: x + y</code> | Lambda function | |
| <code>def foo(first: int, *rest: tuple) -> None:</code> | Variable function args (rest is a tuple) | Call as <code>foo(1, 2, 3)</code> |
| <code>def foo(first: int, **options: dict) -> None:</code> | Variable function args (options is a dict) | Call as <code>foo(1, sec=2, third=3)</code> |

Lists

| | | |
|---|--|--------------------|
| <code>mylist = []</code> | | |
| <code>mylist.append()</code> | | |
| <code>for x in mylist:</code> | | |
| <code>if elem in mylist:</code> | | |
| <code>word_len = [len(words) for word in words if word != " something "]</code> | | List comprehension |
| <code>mylist[start :stop: step]</code> | | List slice |

Modules

| | | |
|--|-------------------------------------|---|
| <code>import my_module</code> | Expects a <code>my_module.py</code> | Usage: <code>my_module.some_func()</code> |
| <code>from my_module import some_func</code> | Import specific from module | Usage: <code>my_func()</code> |
| <code>from my_module import *</code> | Not recommended | |
| <code>import my_module as mine</code> | Alias | Usage: <code>mine.my_func()</code> |

Each .py file is considered a module

Modules are initialized only once at first encountered import. If imported again somewhere else, it will not be loaded anew. This is why local variables act as singletons.

Instance variables versus Class variables

```
class Dude:
    count: int = 0
    def __init__( self, name: str) -> None:
        self.name = name
        Dude.count = Dude.count + 1
print( Dude.count) # 0
dude = Dude("T haD ude ")
print( Dude.count) # 1
```



Instance variables versus Class variables (cont)

```
> king = Dude("King")
print(Dude.count) # 2
print(dude.count) # 2
print(king.count) # 2
king.count = 666
print(Dude.count) # 2
print(dude.count) # 2
print(king.count) # 666 - king just got an instance variable count that takes precedence Dude.count
```

When no variable can be found with the given name on the instance, a fallback happens to the class variable. However when we assign a value as in `king.count = 2` we are creating an instance variable for that object. This is also called Name Shadowing

Type Annotations

```
from typing import Callable
```

```
from typing import Generator
```

```
None | float
```

Multiple possible types

Python is both a strongly typed and a dynamically typed language. Strong typing means that variables do have a type and that the type matters when performing operations on a variable. Dynamic typing means that the type of the variable is determined only during runtime.

Adding type annotations is a huge advantage for anyone using your modules.

Operators

| | |
|---|--------------------------------------|
| <code>%</code> | Modulo |
| <code>**</code> | Power |
| <code>+</code> | Addition, string concat, list concat |
| <code>" 5" * 3</code> | <code>" 555 "</code> |
| <code>[5] * 3</code> | <code>[5, 5, 5]</code> |
| <code>//</code> | Floor division |
| <code>not, and, or</code> | Logical operators |
| <code>is, is not</code> | Identity operators |
| <code>in, not in</code> | Membership operators |
| <code>&, , ~, ^, >>, <<</code> | Bitwise operators |
| <code>x if condition else y</code> | Ternary operator |

Useful Packages

`dacite` This module simplifies creation of data classes (PEP 557) from dictionaries.

`requests` Requests is a simple, yet elegant, HTTP library.

Main

```
def main():
    print(" Hello World")
if __name__ == '__main__':
    main()
```

`main()` is only called when file is ran directly.

If the file is imported into another module, `main()` is not called.

Closures

```
from typing import Callable
def outer_func(m message: str) -> Callable[[], None]:
    def inner_func() -> None:
        print(message)
    return inner_func
# Calling
do_print = outer_func("Hello world")
print("Now we actually print ...")
do_print()
```

While the function `outer_function()` completed, the `message` was rather preserved but hidden and attached to the code.

Decorators make heavy use of this.

Exception Handling

```
try:
    pass
except Exception as e:
    pass
finally:
    pass
```

`e.add_note("Some info")` Add some extra context info to exception

`raise e or raise` Reraise the exception

`raise NewError from e` Chained exception; indicate that `NewError` was caused by `e`

Nested Functions

```
def outer_func(message: str) -> None:
    def inner_func() -> None:
        print(message)

    inner_func()
```

Nested functions can access variables of enclosing scopes, but they are readonly. Can be circumvented using `nonlocal` keyword.



By BioBoost
cheatography.com/bioboost/

Not published yet.
Last updated 6th June, 2026.
Page 3 of 5.

Sponsored by [CrosswordCheats.com](https://crosswordcheats.com)
Learn to solve cryptic crosswords!
<http://crosswordcheats.com>

Loops

`for x in range(5):` range returns iterator

`break, continue` Loop control

`while, for:` Alternative if loop condition false from beginning
`pass`
`else:`

Classes

```
class Dude:
    def __init__(self, name: str) -> None:
        self.name = name
    def introduce(self) -> str:
        return f"Hello, I am {self.name}"
my_object = Dude("Thaddeus")
print(my_object.introduce())
```

Leftovers

`"", None, []` and `{}` These are all considered False in a logical expression

Input / Output

`print(" Hello, %s" % name)` C++ like `printf()`
`print(f"Hello, {name}")` In-place expressions
`astring = input()` Read till newline
`anumber = int(input())` Expects exactly 1 int
`a, b = map(int, input().split())` Expects exactly 2 ints
`array = input().split()`

Sets

```
a = set(["a", "b"])
a.intersection(b)
a.difference(b)
a.union(b)
```

No duplicate entries

JSON

```
import json
json_obj = json.loads(json_string) - Convert string to data object
json_string = json.dumps(json_obj) - Convert data object to string
```

Python supports a similar data serialization method called `pickle` which can be used in exactly the same way.

Code Introspection

`help()` Show help on class, function, method, ...
`dir()` Return all the properties and methods, even built-in properties which are default for all object.
`hasattr()` Returns True if the specified object has the specified attribute
`id()` Returns a unique id for the specified object.
`type()` Returns the type of the specified object
`repr()` Returns the canonical string representation of the object.
`callable()` Return whether the object is callable (i.e., some kind of function).
`issubclass(cls, cls2)` Return whether 'cls' is derived from another class or is the same class.
`isinstance(obj, cls)` Return whether an object is an instance of a class or of a subclass thereof.

Remember that in python everything is an object.



Dictionaries

```
book = { "hello": 55 }           Initialize
book["test"] = 22               Assign element
for key,value in book.items():
del book["test"]               No return value
book.pop("test")               Returns value
if hello in book:
```

Key / Value storage. Key can also be anything

Packages

Needs `__init__.py` file

```
__all__ = [ "public_module" ]  Override exports and
                                keep certain modules
                                internal
```

A directory that gets its own namespace containing modules and other packages.

Regular Expressions

```
import re
pattern = re.compile(r"...")
```

Todo: add search, replace, matching, ...

Generators

```
import random
from typing import Generator
def lottery() -> Generator[int, None, None]:
    for i in range(6):
        yield random.randint(1, 43)
        yield random.randint(1, 20) # Joker
# Usage:
for number in lottery():
    print(number)
```

Functions that returns an iterable set of items one at a time (yield).

Debugging

```
s  Execute the current line, stop at the first possible occasion
c  Continue execution, only stop when a breakpoint is encountered.
h  Print help
p  Print something like a variable
```

Just add `breakpoint()` somewhere in code and when execution comes to this point you will be dropped to `pdb`, an interactive source code debugger.

Decorators

```
def simple_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before the function call")
        func(*args, **kwargs)
        print("After the function call")
    return wrapper
@simple_decorator
def greet(name: str):
    print(f"Hello there {name} !")
greet("King")
# @decorator replaces this:
decorate = simple_decorator(greet)
decorate("King")
```

A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure. Decorators are typically applied to functions, and they play a crucial role in enhancing or modifying the behavior of functions.

When you create a decorator, the wrapper function (inside the decorator) is a closure. It retains access to the function being decorated and any additional state or arguments defined in the decorator function.

