

### Functions

```
def myfunction(first: int, second: str) -> float:
```

```
sum: Callable[[int, int], int] = lambda x, y: x + y
```

Lambda function

```
def foo(first: int, *rest: tuple) -> None:
```

Variable function args (rest is a tuple). Calling: `foo(1, 2, 3)`

```
def foo(first: int, **options: dict) -> None:
```

Variable function args (options is a dict). Call as `foo(1, sec=2, third=3)`

### 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.

### Loops

```
for x in range([start, ] end [, end, step optional. Range = [start, end[
step]]):
```

`break, continue` Loop control

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

`else:`

### Lists

```
mylist = []
```

Initialization

```
.append()
```

Add element to list

```
for x in mylist:
```

Foreach

### Lists (cont)

```
if elem in mylist:
```

Check if element in list

```
mylist[start:stop:step]
```

List slice (all parts optional)

```
lengths = [len(w) for w in words if w != "-
skip"]
```

List comprehension

### Sets

```
a = {"a", "b"}
```

Initialization

```
a.intersection(b)
```

Elements in a and b

```
a.difference(b)
```

Elements in a but not in b

```
a.union(b)
```

All elements

Sets do not have duplicate entries.  
Allow for easy combine operations.

### Closures

```
from typing import Callable
def outer_function(message: str) -> Callable[[],
None]:
    def inner_function() -> None:
        print(message)
    return inner_function
# Calling
do_print = outer_function("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.

### Type Annotations

```
from typing import Callable
```

```
from typing import Generator
```



### Type Annotations (cont)

*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.

### Modules

*import my\_module* Expects a *my\_module.py*. Usage: *my\_module.some\_func()*

*from my\_module import some\_func* Import specific from module. Usage: *my\_func()*

*from my\_module import \** Not recommended. Imports everything.

*import my\_module as mine* Alias. Usage: *mine.my\_func()*

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.

### Data Types

*int, float, str* Basic data types

*True, False* Boolean

*[1, 2, 3]* List

*("hello", "world")* Tuple

*{"key": "value"}* Dictionary

*{"first", "second"}* Set <sup>1</sup>

<sup>1</sup> Create an empty set with *set()*

### Packages

Needs *\_\_init\_\_.py* file

*\_\_all\_\_ = [ "public\_module" ]* Override exports and keep certain modules internal

A directory that get's its own namespace containing modules and other packages.

### 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("T haD ude ")
print( my_object.introduce())
```

### 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.

### Exception Handling

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



### Leftovers

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

### Useful Packages

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

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

### 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`).

### 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.

### 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.

### Regular Expressions

```
import re
```

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

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

### 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()* Return whether 'cls' is derived from another class or is the same class.

*isinstance()* Return whether an object is an instance of a class or of a subclass thereof.

Remember that in python everything is an object.

### 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.



### Dictionaries

<code>book = { "hello": 55 }</code>	Initialize
<code>book["test"] = 22</code>	Store value at key
<code>for key,value in book.items():</code>	Foreach
<code>del book["test"]</code>	Delete without return value
<code>elem = book.pop("test")</code>	Pop with return
<code>if hello in book:</code>	Check if key present

Key / Value storage. Key can also be anything.

### Input / Output

<code>print("Hello, %s" % name)</code>	Print formatting
<code>print(f"Hello, {name}")</code>	In-place expressions
<code>astring = input()</code>	Read till newline
<code>anumber = int(input())</code>	Expects exactly 1 int
<code>a, b = map(int, input().split())</code>	Expects exactly 2 ints
<code>array = input().split()</code>	

### Operators

<code>a % b</code>	a module b
<code>a ** b</code>	a <sup>b</sup>
<code>a + b</code>	Addition, string concat, list concat
<code>"5" * 3</code>	"555"
<code>[5] * 3</code>	[5, 5, 5]
<code>a // b</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>&amp;,  , ~, ^, &gt;&gt;, &lt;&lt;</code>	Bitwise operators

### Operators (cont)

<code>x if condition else y</code>	Ternary operator
<code>a op= b</code>	Shortened assignment <sup>1</sup>

<sup>1</sup> op can basically be any operator

### 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.c ount) # 0
dude = Dude("T haD ude ")
print( Dude.c ount) # 1
king = Dude("K ing ")
print( Dude.c ount) # 2
print( dud e.c ount) # 2
print( kin g.c ount) # 2
king.count = 666
print( Dude.c ount) # 2
print( dud e.c ount) # 2
print( kin g.c ount) # 666
```

*king* just got an instance variable *count* that takes precedence over *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

### Exception Handling cont.

<code>e.add_note("Some info")</code>	Add some extra context info to exception
<code>raise e or raise</code>	Reraise the exception
<code>raise NewError from e</code>	Chained exception; indicate that NewError was caused by e

