Functions

```
def my_fun cti on( first: int, second: str) -> float:
```

`sum: Callab le[ [int, int], int] = lambda x, y: x + y`	Lambda function	
`def foo(first: int, *rest: tuple) -> None:`	Variable function args (rest is a tuple)	Call as `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)`

Lists

```
mylist = []
mylist.ap pend()
for x in mylist:
if elem in mylist:
```

`word_len = [len(w ords) for word in words if word != " som eth ing "]`	List comprehension
`mylist [st art :st op: step]`	List slice

Modules

`import my_module`	Expects a `my_mod ule.py`	Usage: `my_mod ule.so me_ func()`
`from my_module import some_func`	Import specific from module	Usage: `my_func()`
`from my_module import *`	Not recommended	
`import my_module as mine`	Alias	Usage: `mine.m y_f unc()`

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:
                sel f.name = name
                Dud e.count = Dude.count + 1
print( Dud e.c ount) # 0
dude = Dude("T haD ude ")
print( Dud e.c ount) # 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

`%`	Modulo
`**`	Power
`+`	Addition, string concat, list concat
`" 5" * 3`	`" 555 "`
`[5] * 3`	`[5, 5, 5]`
`//`	Floor division
`not`, `and`, `or`	Logical operators
`is`, `is not`	Identity operators
`in`, `not in`	Membership operators
`&`, `|`, `~`, `^`, `>>`, `<<`	Bitwise operators
`x if condition else y`	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():
    pri nt( " Hello World")
if __name__ == '__mai n__':
    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_ fun c(m essage: str) -> Callab le[[],
None]:
       def inner_ func() -> None:
                pri nt( mes sage)
        return inner_func
# Calling
do_print = outer_ fun c("Hello world")
print( "Now we actually print ...")
do_print()
```

While the function `outer_ fun ction()` 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_ not e("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:
        pri nt( mes sage)

    inn er_ func()
```

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

Loops

`for x in range(5):`	`range` returns iterator
`break`, `continue`	Loop control

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

Classes

```
class Dude:
       def __init __( self, name: str) -> None:
                sel f.name = name
       def introd uce (self) -> str:
                return f"Hello, I am {self.n am -
e}"
my_object = Dude("T haD ude ")
print( my_ obj ect.in tro duce())
```

Leftovers

`" "`, `None`, `[]` and `{}`	These are all consider `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(in put())`	Expects exactly 1 int
`a, b = map(int, input( ).s plit( ))`	Expects exactly 2 ints
`array = input( ).s plit()`	

Sets

```
a = set(["a ", " b"])
a.inte rse cti on(b)
a.diff ere nce(b)
a.union(b)
```

No duplicate entries

JSON

```
import json
```

`json_obj = json.l oad s(j son _st - ring)`	Convert string to data object
`json_s tring = json.d ump s(j son - _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).
`issubc lass( )`	Return whether 'cls' is derived from another class or is the same class.
`isinst ance( )`	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 = { " hel lo": 55 }`	Initialize
`book["t est "] = 22`	Assign element
`for key,value in book.i tems():`	
`del book["t est "]`	No return value
`book.p op( " tes t")`	Returns value
`if hello in book:`	

Key / Value storage. Key can also be anything

Packages

Needs `__init __.py` file

`__all__ = [ " pub lic _mo dul e" ]`	Override exports and keep certain modules internal

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

Regular Expressions

```
import re
pattern = re.com pil e(r "..." )
```

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

Generators

```
import random
from typing import Generator
def lottery() -> Genera tor [int, None, None]:
       for i in range(6):
                yield random.ra ndi nt(1, 43)
        yield random.ra ndi nt(1, 20) # Joker
# Usage:
for number in lottery():
        pri nt( 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 `breakp oint()` 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 wrappe r(* args, **kwargs):
                pri nt( " Before the function
call")
                fun c(* args, **kwargs)
                pri nt( " After the function
call")
        return wrapper
@simpl e_d eco rator
def greet( name: str):
        pri nt( f"Hello there {name} !")
greet( " Kin g")
# @decorator replaces this:
decorate = simple _de cor ato r(g reet)
decora te( " Kin g")
```

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.