Agenda

- ❑ Functional Programming
- ❑ Unpacking arguments - *args, **kwargs
- ❑ What is a decorator and how to use it
- ❑ How to create user-defined decorator and use it
- ❑ Static Typing vs. Duck Typing
- ❑ Variable Annotations
- ❑ MyPy library
- ❑ Function Annotations
- ❑ Generics using typing module
- ❑ Shallow Copy vs. Deep Copy - copy module
- ❑ More collections – namedtuples, defaultdict, Counter, ChainMap etc.
- ❑ Assertions for testing
- ❑ Log API
- ❑ What's new in Python 3.9

Functional Programming

- ❑ A programming paradigm in which programs are constructed using function
- ❑ In Functional programming, functions only take inputs and produce outputs, and don’t have any internal state and side effects
- ❑ Functions are treated as first-class citizens (just like any other object)
- ❑ Functions can be assigned to variables, passed as arguments, and returned from other functions
- ❑ Python supports functional programming and also object oriented programming

Functions as first-class citizens

- ❑ A function can be assigned to a variable
- ❑ A function can be passed to another function as parameter
- ❑ A function can be defined inside another function
- ❑ A function can be returned from another function

Assign function to variable

```
def process():
    print("Function process()")

p = process

p()
process()
```

Function as parameter

```
def operation(func, a, b):
    return func(a, b)

def add(n1, n2):
    return n1 + n2

def mul(n1, n2):
    return n1 * n2


print(operation(add, 10, 20))
print(operation(mul, 10, 20))
```

Function returning function

```
def log_to_file(message):
    print('Logging to file :', message)

def log_to_screen(message):
    print('Logging to screen :', message)

def logger(target):
    if target == 'screen':
        return log_to_screen
    else:
        return log_to_file

log = logger('screen')
log('Testing!')
```

Unpacking arguments - *args and **kwargs

❑ *args represents a set of positional arguments

❑ **kwargs represents a set of keyword arguments

❑ These special arguments are used when a function calls another function

❑ Arguments are unpacked using *args and **kwargs

*args and **kwargs example

```python
def call_func(func, *args, **kwargs):
    func(*args, **kwargs)

def fun1(x, y):
    print("Function 1")

def fun2(name, email):
    print("Function 2")

# 10 and 20 will be in *args
call_func(fun1, 10, 20)

# name and email in **kwargs
call_func(fun2, name='Abc', email='abc@gmail.com')
```

Decorator

❑ Decorator is a function that wraps another function and modifies its behavior

❑ Decorator internally returns a function, which is called when decorated function is invoked

❑ The function that is returned by decorator invokes decorated function and does pre and post process

❑ Associate a decorator with a function using @ followed by name of the decorator (name of the decorator function)

Predefined Decorators

- ❑ @staticmethod
- ❑ @abstractmethod
- ❑ @classmethod
- ❑ @property

Decorator Example

```
def log(func):
    def wrapper_function():
       # pre-process
       func() # Call decorated function
       # post-process
    return wrapper_function

@log
def hello():
    print('Hello Python')


hello()  # Call function that is decorated
```

Decorator Function That Receives Parameters

```
def log(func):
    def decorator_function(*args, **kwargs):
        print('Calling ', func.__name__)
        print('Arguments :', *args, **kwargs)
        func(*args, **kwargs)
        print('Completed', func.__name__)

    return decorator_function

@log
def hello(name):
    print('Hello', name)

hello('Decorators')  # Call function that is decorated
```

Decorator With Parameter

```
def delay(seconds=10):
    def delay_outer_function(func):
        def delay_inner_function(*args, **kwargs):
            print(f"Waiting for {seconds} seconds!")
            time.sleep(seconds)
            func(*args, **kwargs)

        return delay_inner_function

    return delay_outer_function
```

```
@delay(seconds=5)
def printing():
    for n in range(1, 10):
        print(n, end=' ')
```

```
@delay()
def printing_reverse():
    for n in range(9, 0, -1):
        print(n, end=' ')
```

Variable Annotations

```
a : int = 10

a = "abc"

Print(__annotations__)  # Annotations related to module
```

```
{'a': <class 'int'>}
```

MyPy Library – Optional Static Typing For Python

- ❑ MyPy is third party library to be installed using PIP
- ❑ It is a static type checker for Python
- ❑ It can type check your code that is associated with annotations and find common bugs
- ❑ Annotations add no overheads at runtime, they are treated as comments by Python runtime

Function Annotations

- Function annotations allow programmers to provide metadata about different parts of functions
- These expressions are evaluated at compile time (by third party tools) and are completely ignored by runtime
- They are meant to make Python statically typed so that bugs related to data types can be detected early and fixed

Annotation For Function Parameter

```
def func(param: expression=[default], param: expression=[default]):
    ...
```

```
def add(n1: int, n2: int = 0):
    return n1 + n2


print(add(10, 20))
print(add(n1 = 10, n2 = 30))
print(add('abc', 'xyz'))    # Error
```

Annotation For Function Return Value

```
def func(param: expression=[default]) -> expression:
    ...
```

```
def add(n1: int, n2: int) -> int:
    return n1 + n2


print(add(10, 20))
print(add('abc', 'xyz'))    # Error
```

__annotations__ Attribute

❑ Attribute __annotations__ is used to provide annotations related to functions

❑ It returns a dict that contains annotations related to parameters and return type of the function

```
def add(n1: int, n2: int) -> int:
    return n1 + n2


print(add.__annotations__)
```

```
{'n1': <class 'int'>, 'n2': <class 'int'>, 'return': <class 'int'>}
```

Using MyPy

add.py

```python
def add(n1: int, n2: int) -> int:
    return n1 + n2


print(add(10,20))
print(add("abc", "xyz"))
```

```
>mypy add.py
add.py:6: error: Argument 1 to "add" has incompatible type "str";
expected "int"
add.py:6: error: Argument 2 to "add" has incompatible type "str";
expected "int"
Found 2 errors in 1 file (checked 1 source file)
```

Generics using Typing Module

```python
from typing import Dict, List, Tuple

nums : List[int] = [1, 3, 3]
nums.append('abc')                   # Error

t: Tuple[int, float, int] = (1, 20.2, 3)

d: Dict[str, int] = {}
d['k1'] = 20
d['k2'] = 'Xyz'                      # Error
```

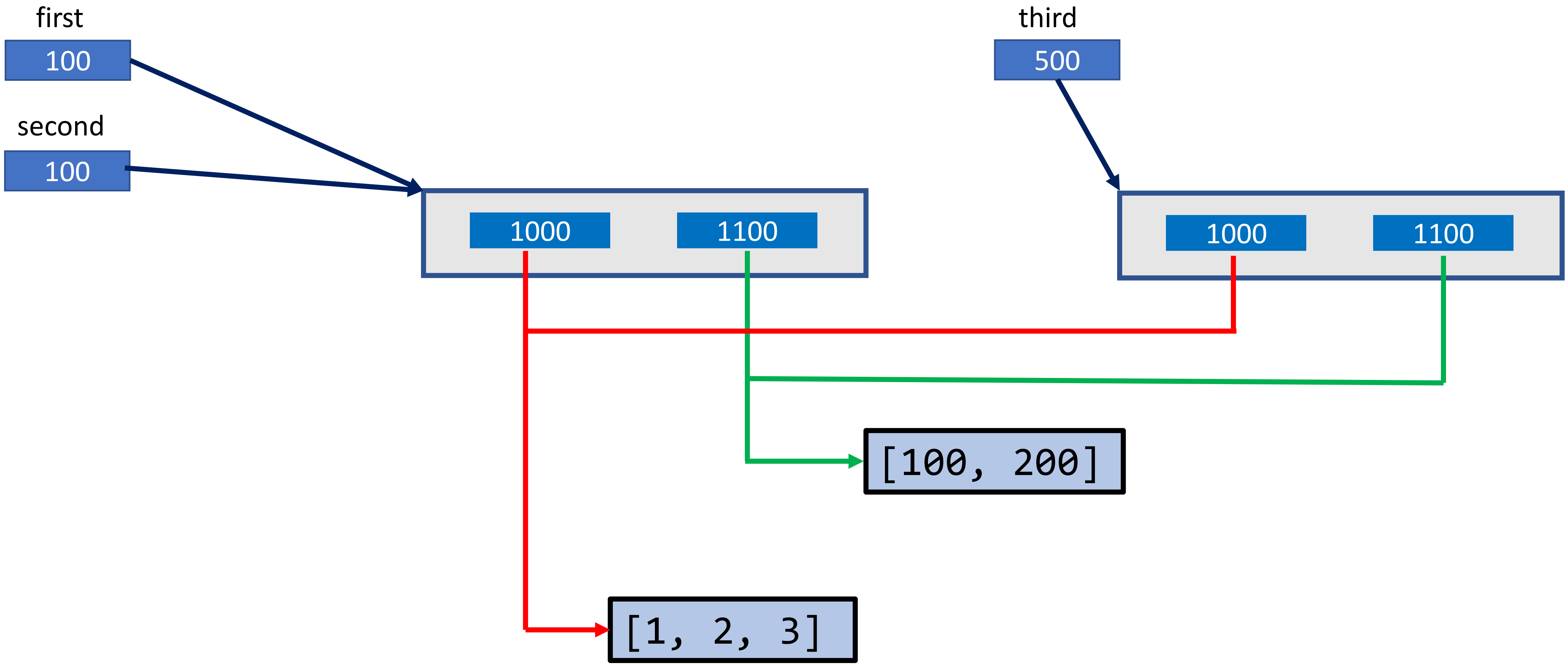
```
>mypy typing_demo.py
typing_demo.py:4: error: Argument 1 to "append" of "list" has incompatible type "str";
expected "int"
typing_demo.py:10: error: Incompatible types in assignment (expression has type "str", target
has type "int")
```

Deep Copy vs. Shallow Copy

- ❑ Shallow copy is where new object is created with references found in original object
- ❑ It is only one level deep
- ❑ Changes to original object can affect new object as only references are copied for child objects
- ❑ Deep copy is where a new object is created and then recursively making new copies of child objects

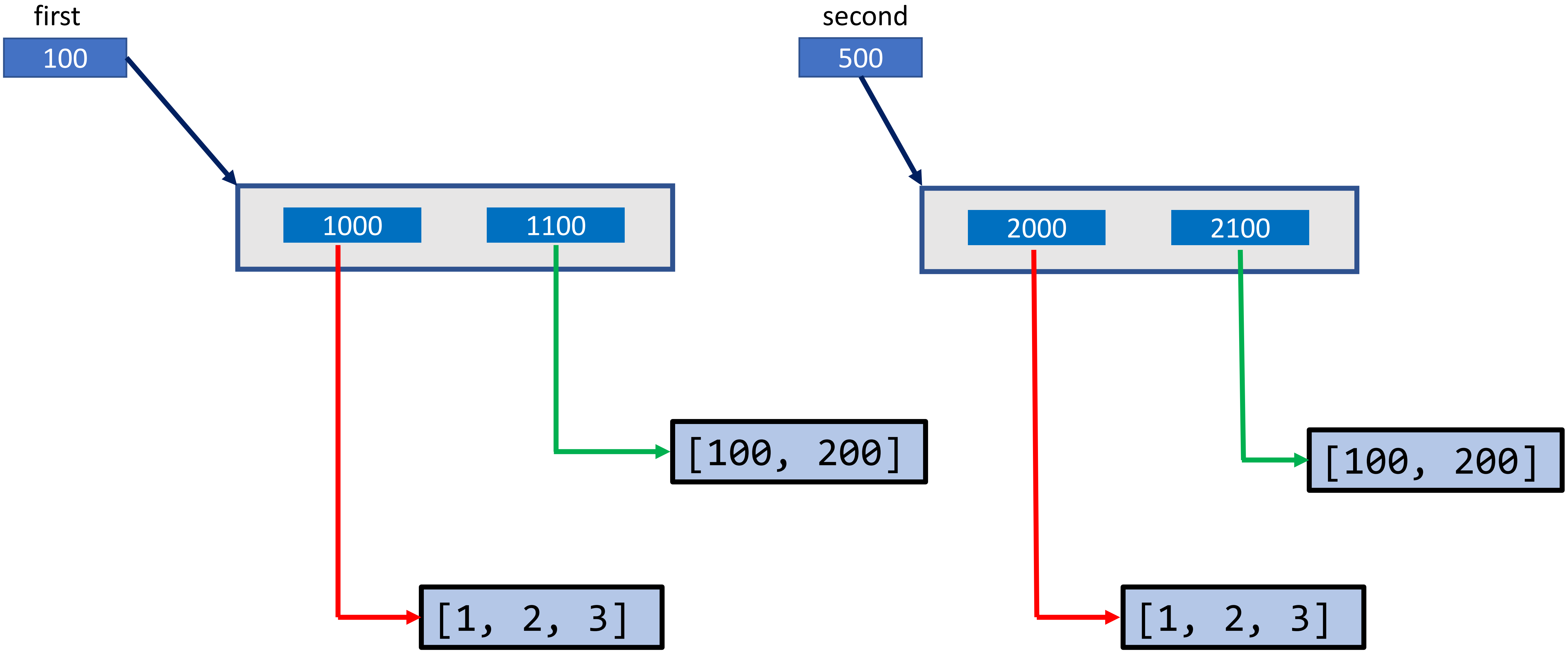
Shallow Copy

```
third = copy.copy(first)
```



Deep Copy

```
second = copy.deepcopy(first)
```



Named Tuples

- Use collections.namedtuple() to create a subclass of tuple with the given name and given fields
- It supports all the features of tuple and in addition, provides fields that are accessible as attributes
- The field_names are a sequence of strings such as ['x', 'y']. Alternatively, field_names can be a single string with each fieldname separated by whitespace and/or commas, for example 'x y' or 'x, y'

```
namedtuple (typename, field_names, *, defaults=None)
```

namedtuple Example

```
from collections import namedtuple

Time = namedtuple('Time', 'hours, minutes, seconds',
                   defaults=[0, 0, 0])

t1 = Time(10, 20, 30)
print(t1, type(t1))
print(t1.hours)
# t1.hours = 10

t2 = Time(minutes=10, seconds=30)
print(t2)
```

Methods and Attributes of namedtuple

_make(iterable)

Class method that makes a new instance from an existing sequence or iterable.

_asdict()

Returns a new dict which maps field names to their corresponding values.

_replace(**kwargs)

Returns a new instance of the named tuple replacing specified fields with new values.

_fields

Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

_field_defaults

Dictionary mapping field names to default values.

Default Dictionary - defaultdict

- Class defaultdict is a subclass of the built-in dict class
- It overrides one method and adds one writable instance variable
- The remaining functionality is the same as for the dict class
- The first argument provides the initial value for the default_factory attribute; it defaults to None

```
defaultdict([default_factory[, ...]])
```

defaultdict Example

```
from collections import defaultdict

marks = defaultdict(list)

marks['Steve'].append(80)
marks['Bill'].append(70)
marks['Steve'].append(87)

for name, ml in marks.items():
    print(name, ml)
```

Counter

- ❑ **A subclass of dict used to count objects**
- ❑ **It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values**
- ❑ **It has dictionary interface except that it returns a zero count for missing items instead of throwing error**

```
Counter([iterable-or-mapping])
```

ChainMap

❑ **Groups multiple dicts to create a single, updateable view.**

❑ **The underlying mappings are stored in a list. That list can be accessed or updated using the *maps* attribute.**

❑ **It incorporates the underlying mappings by reference. So, if one of the underlying mappings gets updated, those changes will be reflected in ChainMap.**

❑ **Supports all of the usual dictionary methods.**

```
ChainMap(*maps)
```

Logging

- ❑ Logging is the means to keep track of events that occur during execution of software
- ❑ The importance of the event is called level
- ❑ Module logging is used to log errors, warnings and informative messages
- ❑ Functions in logging module are used to log
- ❑ Logging level, target, format etc. can be configured using basicConfig() function

Function of logging module

getLogger(name)
Return a logger with the specified name, if name is given, otherwise return root logger.

debug(msg, *args, **kwargs)
Logs a message with log level debug. Parameters are merged into msg object.

info(msg, *args, **kwargs)
Logs a message with log level INFO.

warning(msg, *args, **kwargs)
Logs a message with log level WARNING.

error(msg, *args, **kwargs)
Logs a message with log level ERROR.

basicConfig(**kwargs)

Format	Description
filename	Specifies that a FileHandler be created, using the specified filename.
filemode	If filename is specified, open the file in this mode. Defaults to 'a'.
format	Use the specified format string for the handler.
datefmt	Use the specified date/time format.
level	Set the root logger level to the specified level.

```
logging.basicConfig(level=logging.DEBUG)
```

```
logging.basicConfig(filename="log.txt",
                    format="%(levelname)s:%(asctime)s:%(message)s")
logging.warning("This is a warning!")
```

Logging Levels

Level	When it’s used
DEBUG	Detailed information, typically of interest only when diagnosing problems.
INFO	Confirmation that things are working as expected.
WARNING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.
ERROR	Due to a more serious problem, the software has not been able to perform some function.
CRITICAL	A serious error, indicating that the program itself may be unable to continue running.

Assertions

- Assertions are used to test whether code is run according to our expectations.
- If an assertion is true then it does nothing, if it is false then AssertionError is raised.
- The assert keyword is used to include assertions in Python.
- Assertions may be ignored at runtime, so do not use assertions for conditions that are crucial to functionality of the application. Instead use assertions only for testing.
- Use –O with python.exe to turn on Optimization and turn off assertions.

```
assert <boolean_expression>, message
```

```
assert iseven(10), 'Should return True but returns False'
```

Testing

- ❑ This is where you test your application to ensure every feature is working correctly
- ❑ It can be done manually or automatically
- ❑ Testing must be done every time you change code
- ❑ When manual testing is done, every feature is manually tested by a human
- ❑ When automatic testing is done, every feature is tested with a script
- ❑ Unit test is where you test one component of your application at a time
- ❑ Integration test is where you test all components put together

Module unittest

- ❑ This is Python standard library for testing
- ❑ Inspired by JUnit
- ❑ It is provided through unittest module
- ❑ Test case class is created by subclassing unittest.TestCase
- ❑ It expects you to put your test code as methods in class
- ❑ These methods use assertion method provided by TestCase class of unittest
- ❑ Names of the method must start with test

UnitTest

```
import unittest

class TestIsSorted(unittest.TestCase):
    def test_list(self):
        self.assertTrue(issorted([1, 2, 3]), "Testing a list")

unittest.main()
```

New Features of Python 3.9

❑ New functions in math module – lcm(), gcd()

❑ Dictionary update using |= and |

❑ Generic list and dict

❑ String methods - removePrefix(), removeSuffix()

❑ Adding context specific metadata – Annotated, get_type_hints() of typing module