A Beginner's Guide to Class Construction in Python

Satya Komatineni, 2/21/24 (version 5) [started 2/1/24]

Location: <https://github.com/SatyaKomatineni/articles-repo/blob/master/classes-in-python-a-beginers-guide.docx>

Audience: Fairly experienced Object-oriented programmer in any other language.

Absorption time: 10 pages and about 2 leisurely hours.

Contents

[Abstract 2](#_Toc160433103)

[About Typing in Python First 2](#_Toc160433104)

[Simple Class Construction and Instance Variables 2](#_Toc160433105)

[Encapsulation with Private and Public Methods 3](#_Toc160433106)

[Importance of "self" 4](#_Toc160433107)

[Inheritance and Method Overriding 4](#_Toc160433108)

[Abstract Classes and Method Implementation 5](#_Toc160433109)

[Multiple Inheritance with a Surprise 5](#_Toc160433110)

[Less important for most 6](#_Toc160433111)

[Class Variables and Initialization 6](#_Toc160433112)

[Properties for Access Control 7](#_Toc160433113)

[Static methods Example 7](#_Toc160433114)

[\_\_Call\_\_ method: Objects as functions 8](#_Toc160433115)

[The lowdown on underscores 8](#_Toc160433116)

[A single underscore \_ : 8](#_Toc160433117)

[Double underscores \_\_ : 8](#_Toc160433118)

[Dunder Methods 9](#_Toc160433119)

[Creating singletons and \_\_new\_\_ 9](#_Toc160433120)

[2 Mode Dunder methods: repr(), and str() 10](#_Toc160433121)

[The nature of repr() 10](#_Toc160433122)

[The nature of str() 10](#_Toc160433123)

[Context Managers 10](#_Toc160433124)

[Behavior of With 11](#_Toc160433125)

[A bit about yield and generators 12](#_Toc160433126)

[Generics and Generic classes 13](#_Toc160433127)

[Caution 13](#_Toc160433128)

[How to use existing generics 13](#_Toc160433129)

[How to extend a class from a genericized class 14](#_Toc160433130)

[Making a class Generic 15](#_Toc160433131)

[Flexibility of TypeVar 16](#_Toc160433132)

[Another example: A class with multiple type parameters 16](#_Toc160433133)

[Items not covered yet. 18](#_Toc160433134)

[References 19](#_Toc160433135)

# Abstract

Python has Object-Oriented Programming (OOP). For those familiar with OOP in languages like Java or C++, Python's syntax and dynamic nature present a slightly different paradigm.

This guide starts with foundational concepts.

And progresses to slightly advanced aspects of class design in Python, incorporating direct code examples to clarity.

# About Typing in Python First

Note that types are still only hints in python, and the runtime will not object to run even if the type checkers fail.

Also, if you are using vscode, you may have to set the type checking setting to strict to fully take advantage of types and the static type checking.

# Simple Class Construction and Instance Variables

In Python, defining a class and initializing instance variables is straightforward. The \_\_init\_\_ method (Python's constructor) is used to initialize an object's state, with instance variables declared within:

1. class MyBaseClass:

2. name: str #DO NOT INITIALIZE them here

3. age: int #DO NOT INITIALIZE them here

4.

5. def \_\_init\_\_(self, name: str, age: int):

6. self.name = name

7. self.age = age

This snippet demonstrates defining instance variables (name and age) with type annotations, which are hints to the type of variable but are not enforced by Python at runtime. Type annotations help with readability and can be leveraged by tools for type checking.

**Caution:**

Any variable named outside of “init” or other functions of the class are considered “class” variables. So if you initialize the “name’ and “age” above, they become class variables. If you leave them as mere type hints, that is ok, then they are considered merely type hints and documentation for instance variables.

This is why most python libraries DO NOT declare these instance variables at the class level.

If you do, for readability be careful.

Make it a practice if your intention is to use them as class variables, to name them as “class\_variable\_name”.

So here is an alternative:

1. class MyBaseClass:

2. def \_\_init\_\_(self, name: str, age: int):

3. self.name:str = name

4. self.age: int = age

5.

6. #If you want to print the class

7. def \_\_str\_\_(self):

8. return f"{self.name}, {self.age}"

9.

10. def testBaseClass():

11. x = MyBaseClass("hello",5)

12. log.info(f"{x}")

The slight disadvantage is that your readability is less by doing so.

# Encapsulation with Private and Public Methods

Python uses a naming convention to denote private members: a prefix underscore (\_). While not enforced by the Python runtime, this convention signals to developers that a method or variable is intended for internal use only:

1. # Private method (starts with \_)

2. def \_private\_method(self):

3. print("This is a private method")

4.

5. # Public method

6. def public\_method(self):

7. print("This is a public method")

# Importance of "self"

1. When calling another private method you have to use ***self.\_the\_other\_method()***
2. You don't have to pass the self as an argument while calling that other method.
3. Self is automatically passed as the first argument in such cases of calling another method.
4. However, while defining a method inside a class **"self" MUST be the first argument**.
5. The type-checker or the editor usually tells you this while you are coding.

# Inheritance and Method Overriding

Python supports class inheritance, allowing a class to inherit attributes and methods from another class. Overriding methods in the subclass allows for extending or changing the behavior of the base class methods:

1. class MyDerivedClass(MyBaseClass):

2. gender: str

3.

4. def \_\_init\_\_(self, name: str, age: int, gender: str):

5. super().\_\_init\_\_(name, age) # super init method

6. self.gender = gender

7.

8. # Overriding the private method

9. def \_private\_method(self):

10. super().\_private\_method()

11. print("This is a private method in MyDerivedClass")

12.

13. # Overriding the public method

14. def public\_method(self):

15. super().public\_method()

16. print("This is a public method in MyDerivedClass")

Same named methods in derived classes override the base class methods.

# Abstract Classes and Method Implementation

Moving towards more advanced concepts, abstract classes in Python are defined using the ABC module. Abstract classes cannot be instantiated and require subclasses to provide implementations for abstract methods. You get a type error if a derived base class does not implement an abstract method.

1. from abc import ABC, abstractmethod

2.

3. class MyBaseClass(ABC):

4. @abstractmethod

5. def my\_abstract\_method(self):

6. pass

7.

8. class MySubclass(MyBaseClass):

9. def my\_abstract\_method(self):

10. print("Implemented abstract method in MySubclass")

11.

# Multiple Inheritance with a Surprise

Python allows duplicate methods, although I suspect you want to avoid. See example below:

1. class Base1:

2. def method(self):

3. print("Method of Base1")

4.

5. class Base2:

6. def method(self):

7. print("Method of Base2")

8.

9. class Derived(Base1, Base2):

10. def another\_method(self):

11. print("Another method of Derived")

12.

13. # Creating an instance of Derived

14. d = Derived()

15.

16. # Calling methods: surprise

17. d.method()

18. # This will call the method from Base1,

19. due to the order in the inheritance list

20.

21. d.another\_method()

22. # Calls method defined in Derived

### Less important for most

Python uses a specific rule, known as the C3 linearization, to determine the method resolution order when dealing with multiple inheritance.

This can be inspected using the .\_\_mro\_\_ attribute or the mro() method on the class.

It's a concept to understand when working with multiple inheritance to predict which method will be called when multiple base classes define methods with the same name.

Author’s note: For sanity, I can’t imagine I would want to go there. 😊

# Class Variables and Initialization

Class variables are shared across all instances of a class. Python allows the initialization of class variables in various ways, including using decorators:

1. def initialize\_class\_variables(cls):

2. if not hasattr(cls, 'class\_variable'):

3. cls.class\_variable = 10

4. return cls

5.

6. @initialize\_class\_variables

7. class MyClass:

8. class\_variable = None

9.

10. def \_\_init\_\_(self):

11. pass

12.

13. # Example usage

14. obj = MyClass()

15. print(obj.class\_variable) # Output: 10

This code snippet demonstrates a pattern to ensure that class variables are initialized in a controlled manner, leveraging Python's dynamic capabilities.

Some essentials of class variables

1. Any variable that is initialized outside of methods like “\_\_init\_\_” are class variables.
2. It is merely good practice to name these class variables declared as “class\_variable\_name: type” and then initialize them.
3. If you don’t initialize, and leave them with only type hint, they are considered documentation for instance variables, and ignored by run time.
4. These class variables can be initialized with a function using an annotation function.
5. The invocation of this initialization function is ensured by python imports where it is called only once during the import. I assume it is thread safe. (Please verify for yourself)
6. Python doesn't have java like static block that is guaranteed to be executed only once in a multi-threaded environment.

# Properties for Access Control

Python's property decorators provide a Pythonic way to implement getters, setters, and deleters, allowing for controlled access to an object's attributes:

1. class MyClass:

2. def \_\_init\_\_(self, name):

3. self.\_name = name

4.

5. @property

6. def name(self):

7. return self.\_name

8.

9. @name.setter

10. def name(self, value):

11. self.\_name = value

12.

13. @name.deleter

14. def name(self):

15. del self.\_name

Properties encapsulate the internal representation of an attribute, offering an interface for getting, setting, or deleting it, akin to getters and setters in other languages but with a more intuitive syntax.

Notice how you are invoking these methods.

1. obj.name #call the getter

2. obj.name = "blah" #call the setter

3. del obj.name #call the deleter

Notice how annotations differ for each of these methods above.

# Static methods Example

1. These methods annotated as such are at a class level.
2. @staticmethod annotation
3. They don’t have “self” as an argument.

Example

1. class MathOperations:

2. @staticmethod

3. def add(x, y):

4. return x + y

5.

6. # Using the static method

7. result = MathOperations.add(5, 3)

8. print(result) # Output: 8

# \_\_Call\_\_ method: Objects as functions

1. An object of class with a \_\_call\_\_ method will have like a function.
2. It can have arguments.

Example

1. class Multiplier:

2. def \_\_init\_\_(self, factor):

3. self.factor = factor

4.

5. def \_\_call\_\_(self, x):

6. return self.factor \* x

7.

8. # Create an instance

9. doubler = Multiplier(2)

10.

11. # Use the instance as if it were a function

12. print(doubler(5)) # Output: 10

13. print(doubler(10)) # Output: 20

# The lowdown on underscores

You will see “\_” and “\_\_” often used in python.

Here is a quick summary of some of the known places:

### A single underscore \_ :

1. Is used for ignoring values, as in assigning from a tuple (look up docs)
2. Indicating private or internal variables, or by convention to signify that a name is meant for internal use within modules and classes.

### Double underscores \_\_ :

1. At the beginning of a name (but not at the end) enable name mangling, which helps to make an attribute or method private to its class.
2. **Dunder methods:** Double underscores at both the beginning and end of a name (dunder methods) are reserved by Python for special methods, which allow your objects to implement, override, or interact with Python's built-in behavior.
3. **Example of dunder methods are:**
   1. **\_\_Init\_\_**
   2. \_\_call\_\_
   3. \_\_str\_\_

# Dunder Methods

There is a lot of magic wrapped around these built-in over-ridable methods in python.

These methods address the following categories of needs:

1. Initialization and Construction
2. Representation Methods
3. Comparison Methods
4. Arithmetic and Bitwise Operators
5. Container Methods
6. Attribute Access
7. Callable Objects
8. Context Managers
9. Iterable Objects
10. Numeric Conversion
11. Hashing and Boolean Test
12. Descriptor Protocol
13. Copying

Note: Each of these are interesting concepts from the idea of classes in any language. So by the time you are done with Python you will run into them at some point. Knowing about these early you might have better constructed your code.

# Creating singletons and \_\_new\_\_

Here is an example with the \_\_new\_\_ dunder method. Note, I haven’t looked into the threading aspects of this. Do that research as you might need it.

1. class Singleton:

2. \_instance = None # Keep instance reference

3.

4. def \_\_new\_\_(cls, \*args, \*\*kwargs):

5. if cls.\_instance is None:

6. cls.\_instance =

7. super(Singleton, cls).

8. \_\_new\_\_(cls, \*args, \*\*kwargs)

9. return cls.\_instance

10.

11. # Test the Singleton class

12. obj1 = Singleton()

13. obj2 = Singleton()

14.

15. print(obj1 is obj2) # Output: True

# 2 Mode Dunder methods: repr(), and str()

Here is an example:

1. class Product:

2. def \_\_init\_\_(self, name, price):

3. self.name = name

4. self.price = price

5.

6. def \_\_repr\_\_(self):

7. return f"Product({self.name!r}, {self.price!r})"

8.

9. def \_\_str\_\_(self):

10. return f"{self.name} - ${self.price}"

11.

12. # Creating an instance of Product

13. product = Product("Coffee Mug", 12.99)

14.

15. # \_\_repr\_\_ is used when echoing in a console or using repr()

16. print(repr(product)) # Output: Product('Coffee Mug', 12.99)

17.

18. # \_\_str\_\_ is used when printing or using str()

19. print(product) # Output: Coffee Mug - $12.99

## The nature of repr()

The \_\_repr\_\_ method returns a string that could be used to recreate the object. It's a convention to make it look like a valid Python expression that could be used to recreate the object with the same state.

## The nature of str()

Method returns a string that provides a user-friendly description of the object.

# Context Managers

Used for managing resources with the “with” key word.

Here is an example:

1. class ManagedFile:

2. def \_\_init\_\_(self, filename):

3. self.filename = filename

4.

5. def \_\_enter\_\_(self):

6. self.file = open(self.filename, 'r')

7. return self.file

8.

9. def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

10. if self.file:

11. self.file.close()

12.

13. # Usage

14. with ManagedFile('example.txt') as f:

15. content = f.read()

16. print(content)

Here is accomplishing the same with the interesting “yield” concept of python.

Here this is done with a “function” as opposed to a class.

1. from contextlib import contextmanager

2.

3. @contextmanager

4. def managed\_file(filename):

5. try:

6. f = open(filename, 'r')

7. yield f

8. finally:

9. f.close()

10.

11. # Usage

12. with managed\_file('example.txt') as f:

13. content = f.read()

14. print(content)

15.

## Behavior of With

“with” is designed in the language with a specific behavior in each of the above cases.

The first case is how “with” behaves when it is attached to a class.

The second case is how “with” behaves when it is attached to a generator “only” via a “contextmanager” annotation.

#### With a class

In the first case it is easy to imagine what “with” will result in where it calls the “\_\_enter\_\_” first and assigns its return to the “as” variable and calls “\_\_exit\_\_” in an internally “finally” clause.

Simple enough.

#### With a yield

The second case is a “deliberately” orchestrated language specific.

The function has to be a generator. A generator function is one where there is a “yield”.

Read up on the nuances of “yield” and generators.

The behavior is as if the code block in “with” is fused to the code block between the “yield” and the “eventual” return of the function.

This interaction is facilitated only by the “@contextmanager” annotation.

## A bit about yield and generators

The presence of “yield” in source code of a function automatically makes that function a generator. This is almost like a “constructor” of a class where an object is returned, although here it is a generator object.

So calling that method again will only return another generator object and not the value that the “yield” is returning.

The values you want to get from that function you have to call “next(generator)”

See the code and inline explanations below.

# A method with a yield in it is automatically a generator

1. def somegen():

2. code\_before()

3. yield 1

4. yield 2

5. yield 3

6. code\_after()

7.

8. #First get a Generator

9. gen = somegen()

10.

11. #gen is not 1, 2, or 3

12.

13. #if you call somegen() again

14. gen2 = somegen()

15.

16. #gen and gen2 are not the same

17.

18. #Instead you have to call the next() method

19.

20. v1 = next(gen)

21. assert (v1 == 1)

22.

23. #and the execution is suspended at the first yield.

24.

25. v2 = next(gen)

26. assert (v2 = 2)

27.

28. #Execution after the first yield and suspended after the second yield

29.

30. v3 = next(gen)

31. #Execution start after the second yield and suspended after the third yield

32.

33. v4 = next(gen)

34. #Executes now code\_after() and throws a StopIteration exception

35.

One would think that the “with” construct could be using the next() semantics with a catch and a finally clause.

Looks like that is not the case and its behavior is built in.

Either way result is clear where the order of start, body, and end are guaranteed to be executed keeping the “resource” semantics.

# Generics and Generic classes

Starting in 3.11, the collection classes in Python are genericized.

## Caution

Note however that these types are still only hints, and the runtime will not object to run even if the type checkers fail.

Also, if you are using vscode, you may have to set the type checking setting to strict to fully take advantage of this.

## How to use existing generics

For generics older than 3.5 of Python

1. #need for old generics < 3.5

2. from typing import List

3. def testOldGenericLists():

4. oldlist: List[int] = [1,2,3] # This is an old generics

5. #oldlist.append("hello") #fails type check, runs ok

6. oldlist.append(5) # good

7. print(oldlist)

For newer python releases

1. def testNewGenericLists():

2. newlist: list[int] = [4,5,6] # new one is native

3. #newlist.append("hello") #fails type check, runs ok

4. newlist.append(5) # good

5. print(newlist)

## How to extend a class from a genericized class

The following is an example to show the syntax how this works. The example itself is a poor one. Familiarity with this syntax is useful when you look at larger python frameworks where such inheritance is used.

1. class Person():

2. def \_\_init\_\_(self, name: str):

3. self.name = name

4. def \_\_repr\_\_(self) -> str:

5. return self.name

6.

**7. class Organization(list[Person]):**

8. def \_\_init\_\_(self, name:str, address:str):

9. self.name = name

10. self.address=address

11.

12. # The \_\_dict\_\_ holds the key value pairs of self

13. # such as name, address etc.

14. def \_\_repr\_\_(self) -> str:

15. t = (self.\_\_dict\_\_, super().\_\_repr\_\_())

16. return f"{t}"

17.

18. def testOrgainzation():

19. org = Organization("MyOrg","111, 1st Street, Great City")

20. org.append(Person("name1"))

21. org.append(Person("name2"))

22. org.append(Person("name3"))

23. print(org)

Line 7: Notice how the class “Organization” is extending the genericized “list[Person]” class.

Line20, 21,22: This allows only to add Person objects to the “Organization”.

The output looks like the following:

({'name': 'MyOrg', 'address': '111, 1st Street, Great City'}, '[name1, name2, name3]')

## Making a class Generic

Again, you will see examples in frameworks where a class is stipulated to work with only certain types that you can pass at run time.

In such cases you are not extending a previously genericized class like above.

The way this type of class is set up is different.

Here is an example.

Please note that this is purely for syntax clarification and a real-world useful example likely differs in its function. However, this example should be good enough to decipher.

1. # You have to import

2. # \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

3. from typing import TypeVar, Generic

4.

5. # You have to do this first

6. # \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

7. T = TypeVar ("T", bound=Person)

8. # bound means,

9. # "T" represents all the derived classes of Person

10.

11. # Next

12. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

13. # Use that type "T" as the generic

14. # Organization2 can now be parameterized

15. # for list of things whose type is T

16. #

17. # In other words you can have

18. # 1. an orgaization of people

19. # 2. Or an organization of plants

20. # etc.

21. #

22. class Organization2(Generic[T]):

23. def \_\_init\_\_(self, name:str, address:str):

24. self.name = name

25. self.address=address

26. self.list: list[T] = []

27.

28. def add(self, p: T):

29. self.list.append(p)

30.

31. # The \_\_dict\_\_ holds the key value pairs of self

32. # such as name, address etc.

33. def \_\_repr\_\_(self) -> str:

34. t = (self.\_\_dict\_\_, super().\_\_repr\_\_())

35. return f"{t}"

36.

37.

38. def testOrgainzation2():

39. org = Organization2[Person]

40. ("AnOrg","111, 1st Street, Great City")

41. org.add(Person("name1"))

42. org.add(Person("name2"))

43. org.add(Person("name3"))

44. # org.add("hello") will fail

45. print(org)

46.

Key points:

1. You have import Generic and TypeVar from typing.
2. You must declare a TypeVar.
3. You then must extend the Generic Class with the TypeVar as its parameter.

## Flexibility of TypeVar

A TypeVar allows a type specification that is more precise.

Here is an example:

Some\_name = TypeVar(“Some\_name”, int, str, bound=”SomeBaseClass”)

1. The first argument must match the name of the variable.
2. The series of types, (here: int, str) are the allowed types for the defined type of variable. It is an “or”.
3. The “bound” is the base class.
4. There is another argument around “covariance” and “contravariance” which for now I am not covering. Read the references provided if you need to know about that one.

## Another example: A class with multiple type parameters

Go through the code now, as this should be clear to you now.

1. from baselib import baselog as log

2.

3. """

4. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

5. \* To implement generics, import the following

6. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

7. """

8. from typing import Generic, TypeVar

9.

10. """

11. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

12. \* Define tyype variables

13. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

14. """

15. # Define two type variables, T and U, which can be any types

16. T = TypeVar('T')

17. U = TypeVar('U')

18.

19. """

20. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

21. \* Use type variable to parameterize classes

22. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

23. """

24. # Define a generic class Pair, which hold items of types T and U

25. class Pair(Generic[T, U]):

26. def \_\_init\_\_(self, first: T, second: U):

27. self.first = first

28. self.second = second

29.

30. def get\_first(self) -> T:

31. return self.first

32.

33. def get\_second(self) -> U:

34. return self.second

35.

36. def \_\_repr\_\_(self) -> str:

37. return f'Pair({self.first!r}, {self.second!r})'

38.

39. """

40. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

41. \* Use the new type

42. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

43. """

44. def \_test1():

45. pair1: Pair[int,int] = Pair(5,6)

46. log.ph("Testing int, int pair", pair1)

47.

48. def \_test2() -> Pair[str,str] :

49. return Pair("first", "second")

50.

51. def \_test3():

52. pair: Pair[str,str] = \_test2()

53. log.ph("Testing string, string pair", pair)

54.

55. def \_test4():

56. log.ph1("Broader test")

57. # Pair of an integer and a string

58. pair\_int\_str = Pair(1, "Apple")

59. print(pair\_int\_str)

60.

61. # Pair of a string and a list of floats

62. pair\_str\_list = Pair("Temperatures", [32.5, 31.8, 30.2])

63. print(pair\_str\_list)

64.

65. # Accessing elements

66. print(pair\_int\_str.get\_first()) # Output: 1

67. print(pair\_int\_str.get\_second()) # Output: Apple

68. print(pair\_str\_list.get\_second()) # Output: [32.5, 31.8, 30.2]

69.

70. def \_test():

71. \_test1()

72. \_test3()

73. \_test4()

74.

75. def localTest():

76. log.ph1("Starting local test")

77. \_test()

78. log.ph1("End local test")

79.

80. if \_\_name\_\_ == '\_\_main\_\_':

81. localTest()

Note that you can use the “bound” parameter on types to enforce some base classes if necessary.

# Items not covered yet.

The following ideas are around Generics. For now, these are not covered.

1. Covariance
2. Contravariance

# References

1. Python docs (PEPs – Python Enhancement Proposals, these are like Java JSRs) on typing: <https://peps.python.org/topic/typing/>
2. Read up on TypeVar from typing module of the standard library: <https://docs.python.org/3.11/library/typing.html#typing.TypeVar>
3. PEP 484: Python docs on type hints: <https://peps.python.org/pep-0484/>
4. PEP 646: Python docs on generics: <https://peps.python.org/pep-0646/>