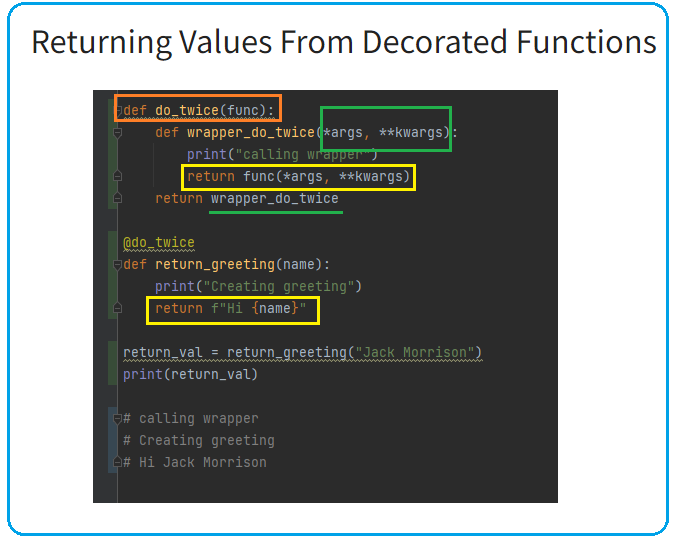
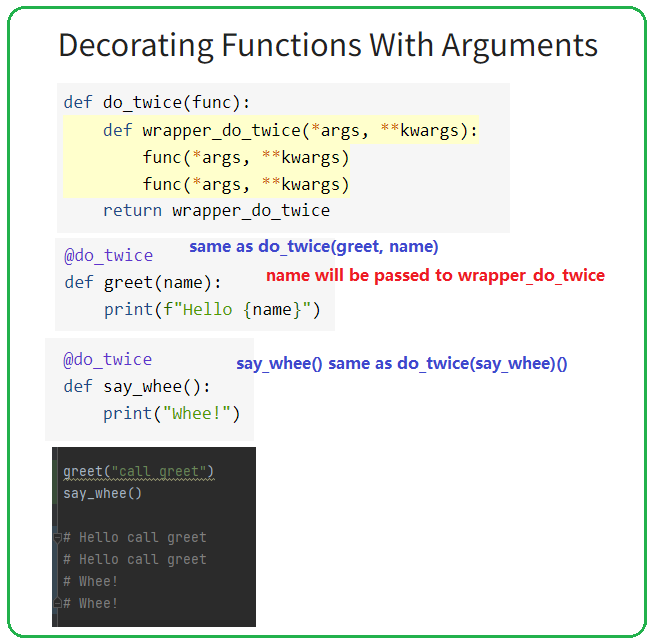
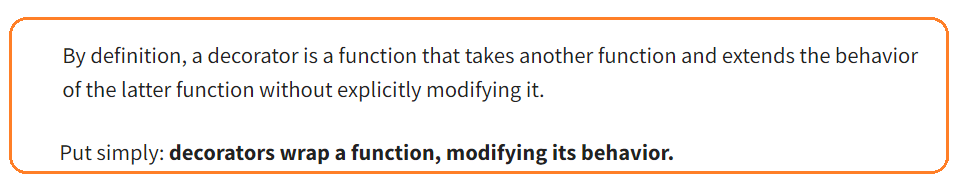
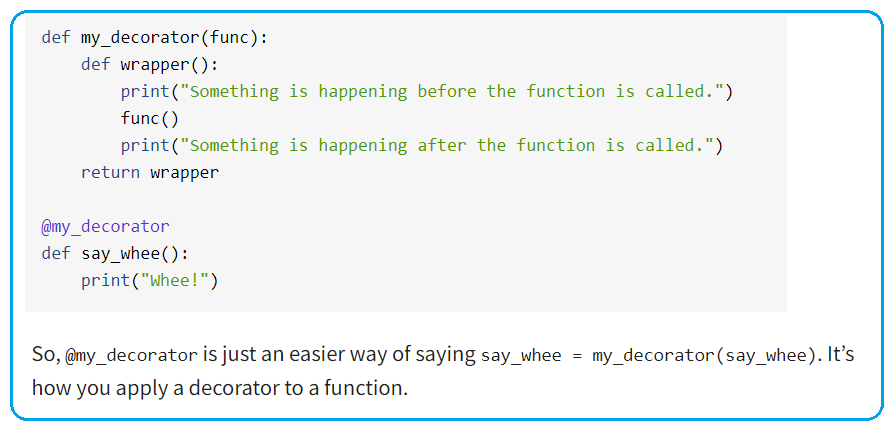
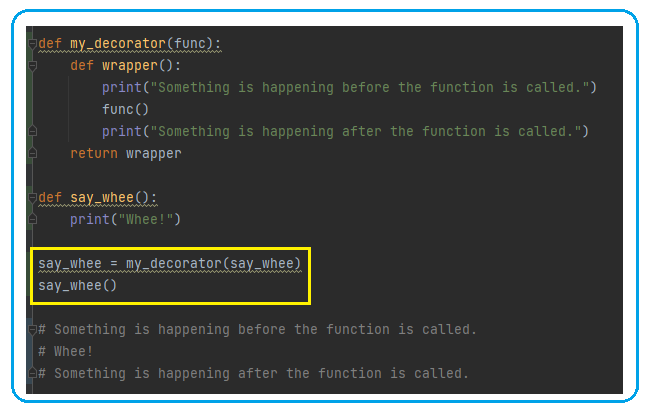
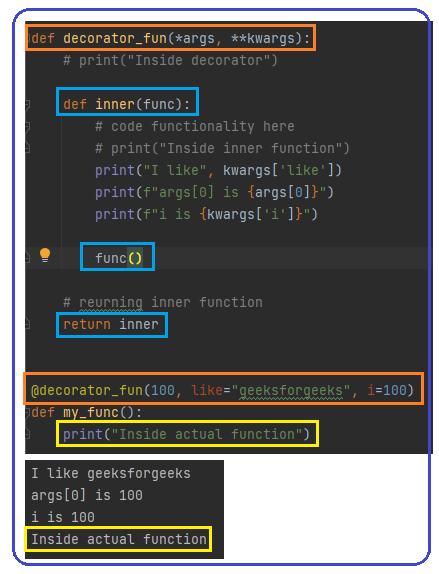
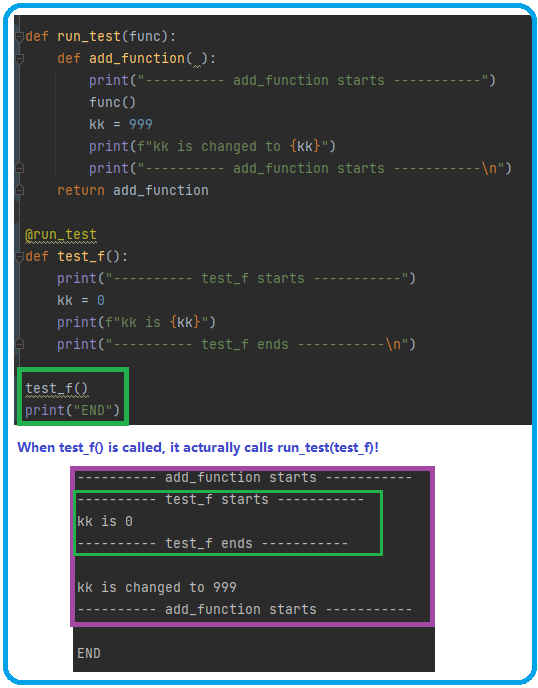
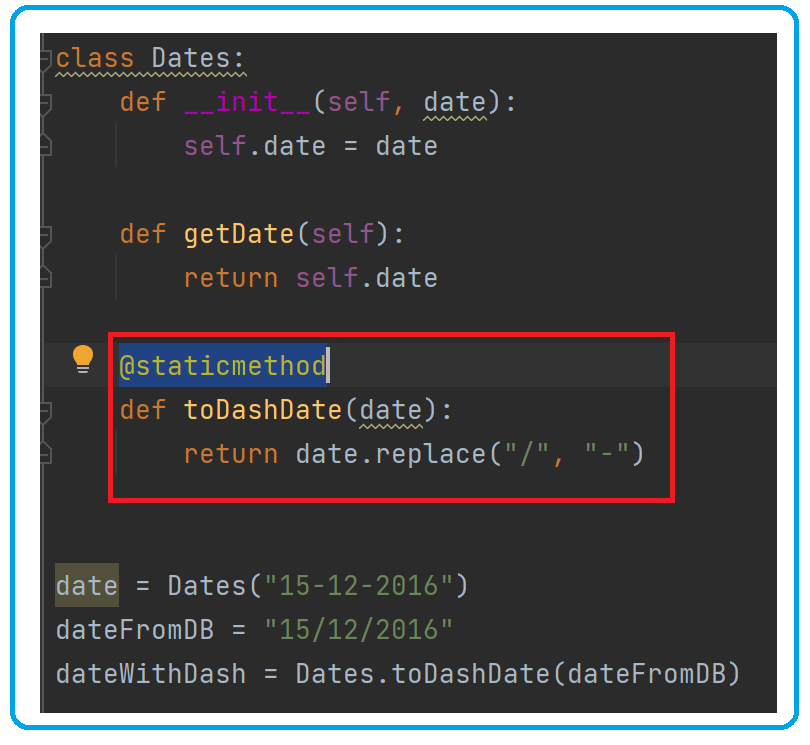
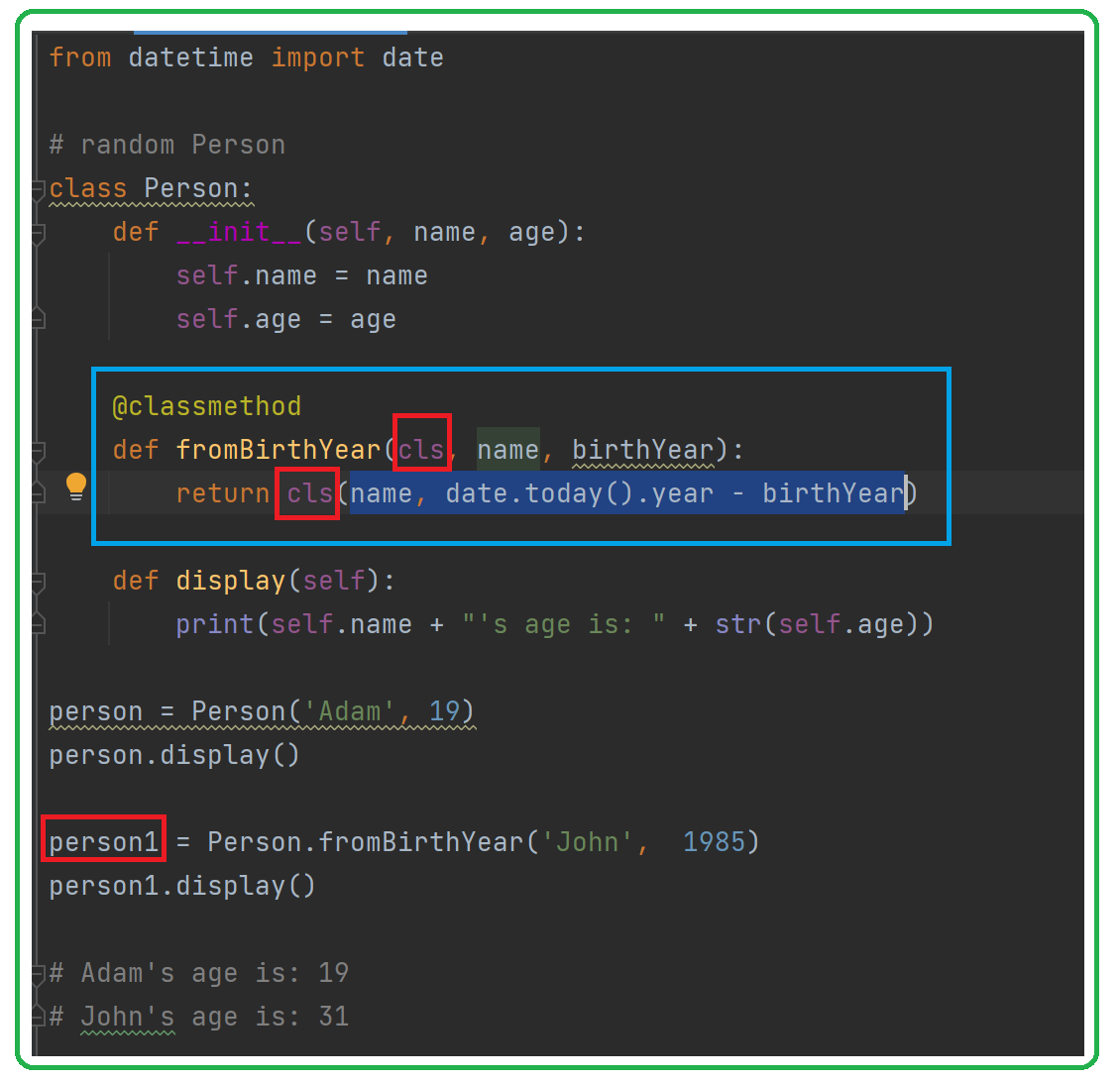
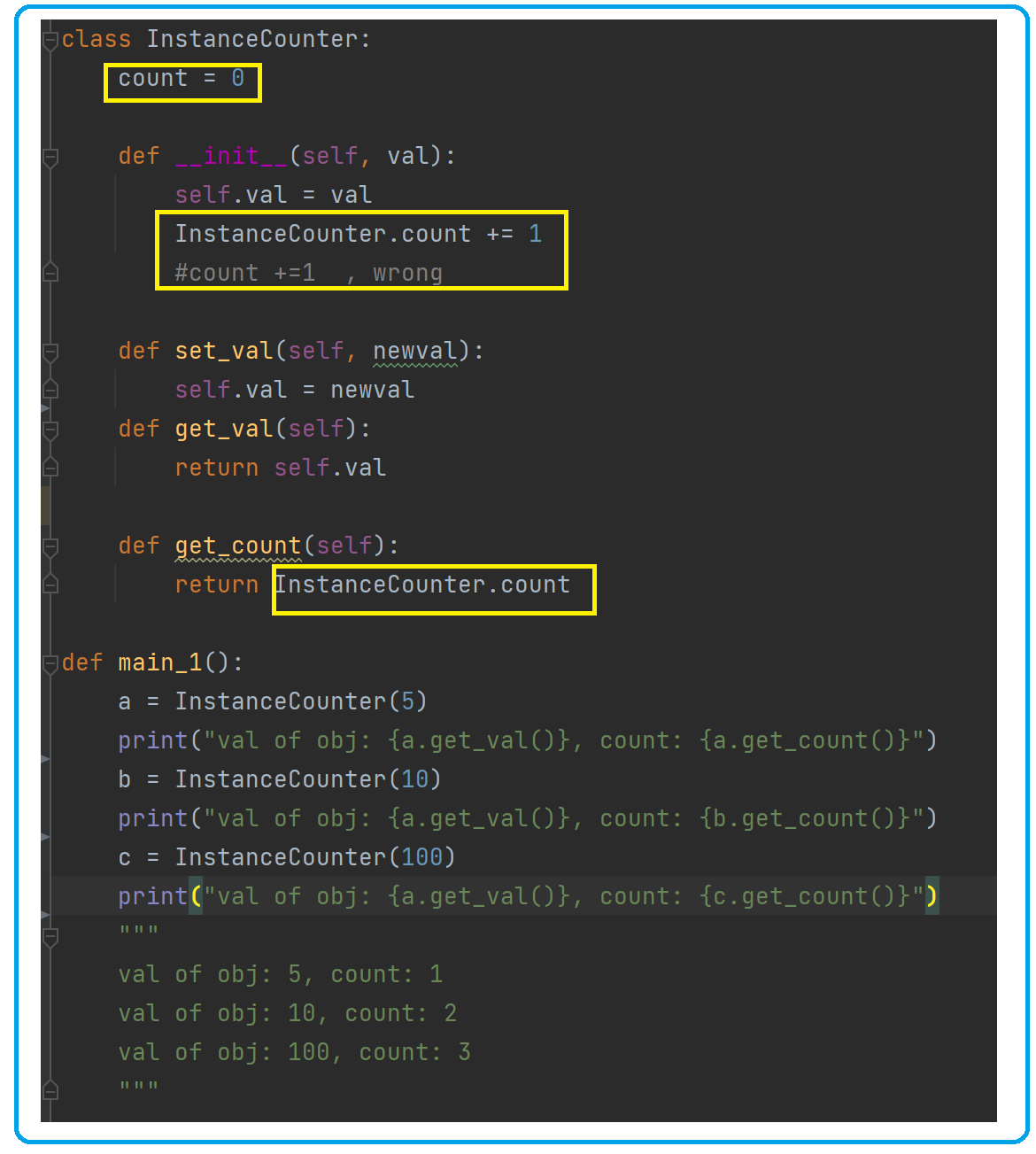
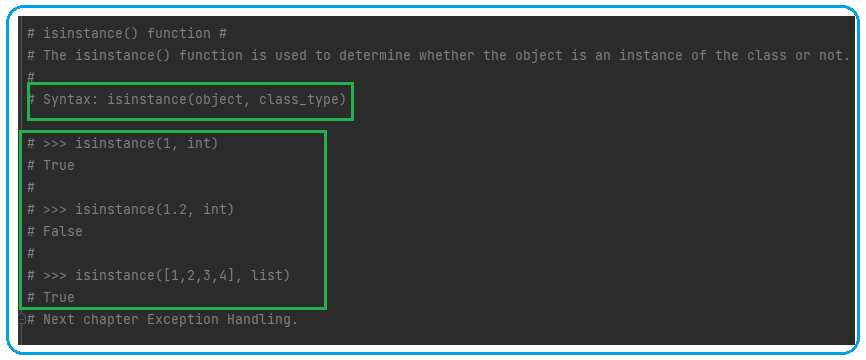
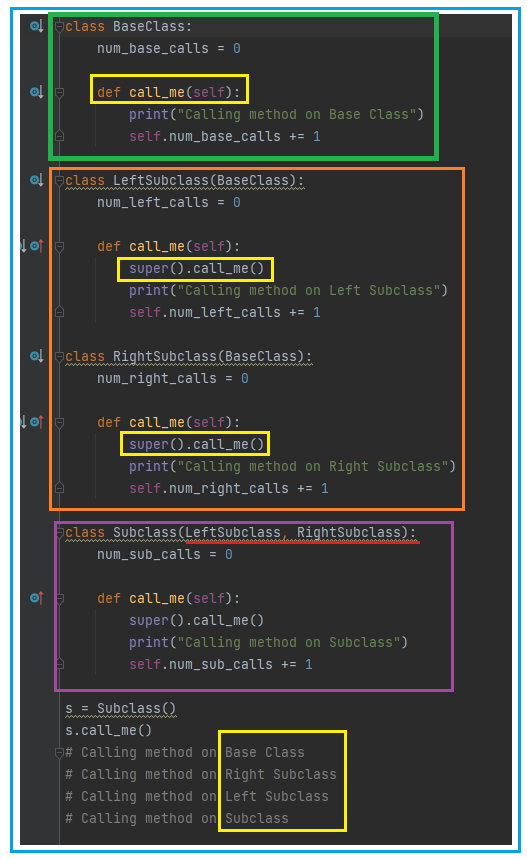
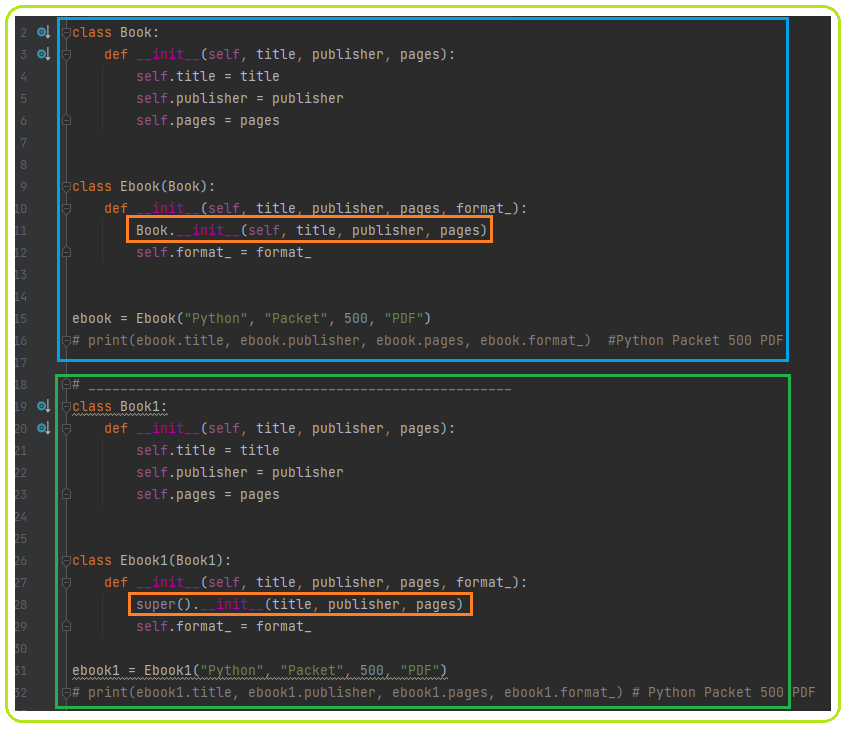
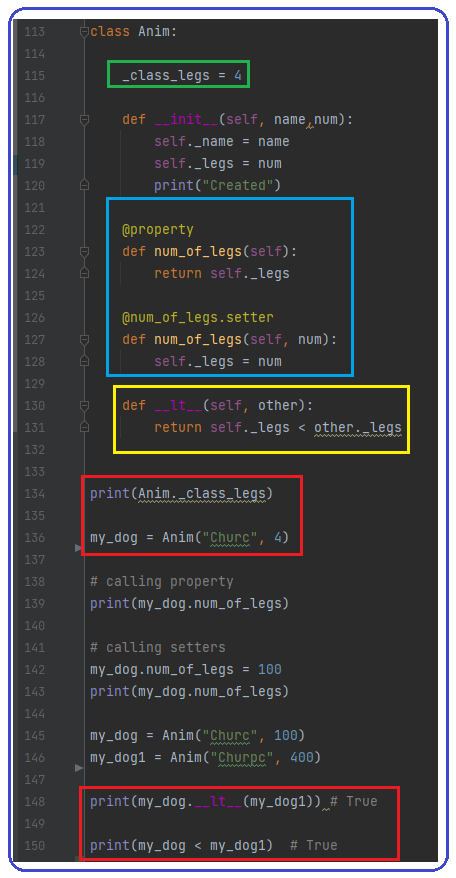
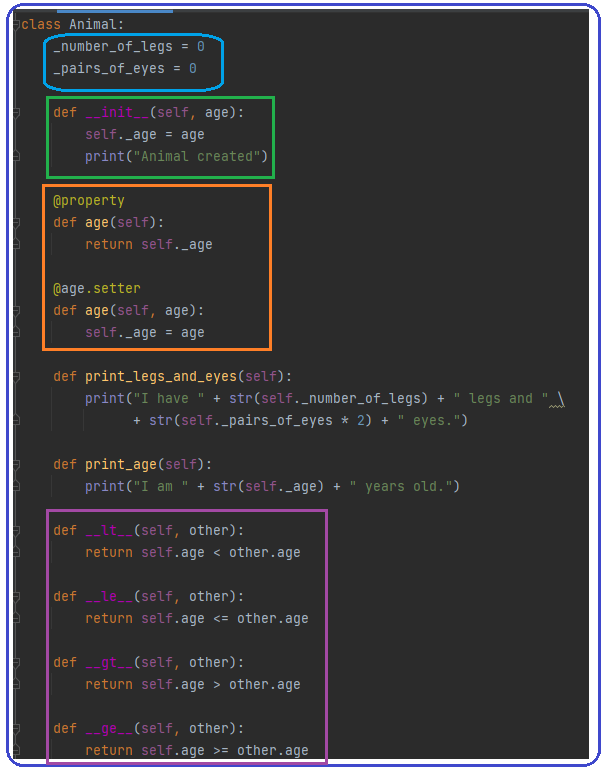
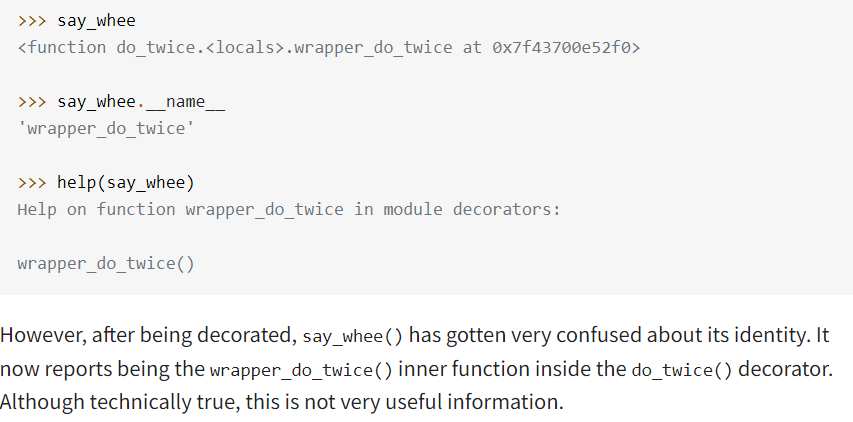
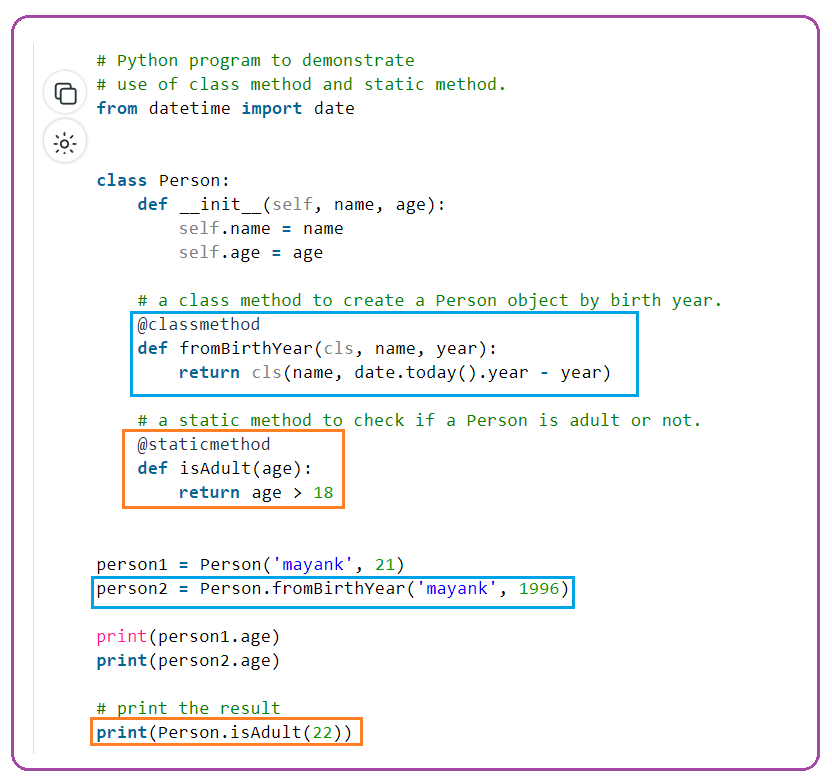
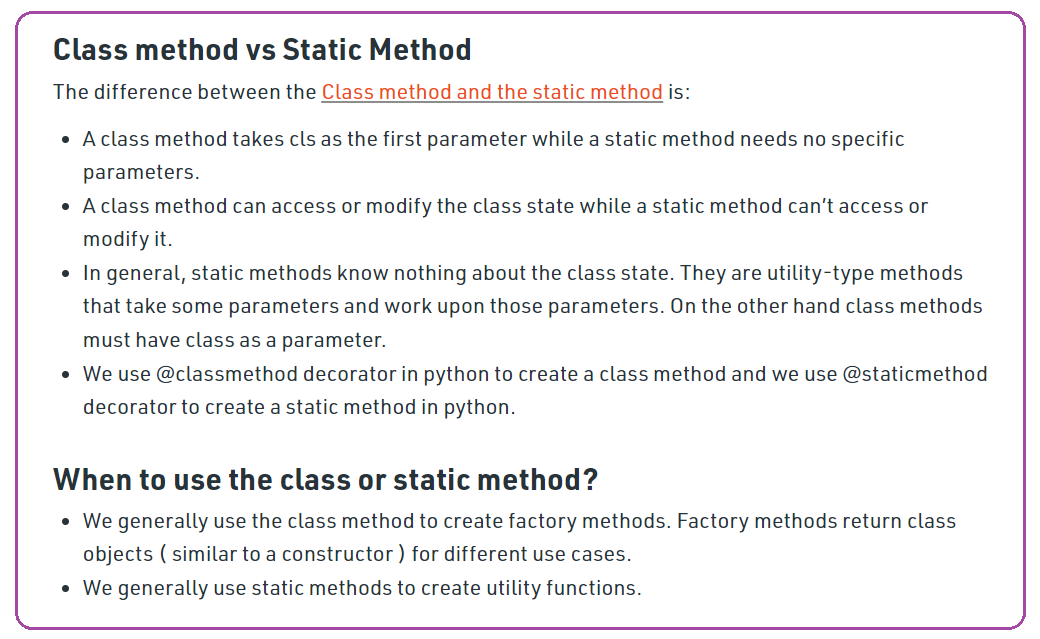
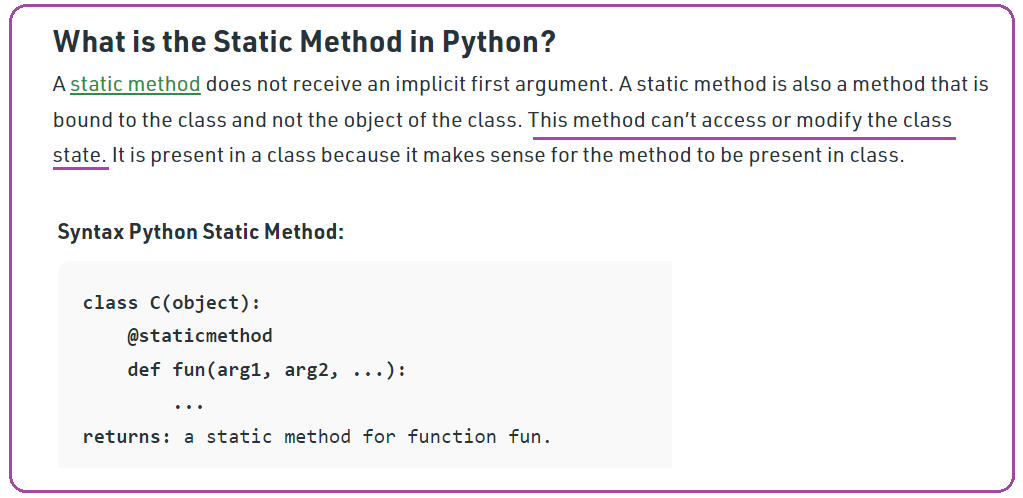
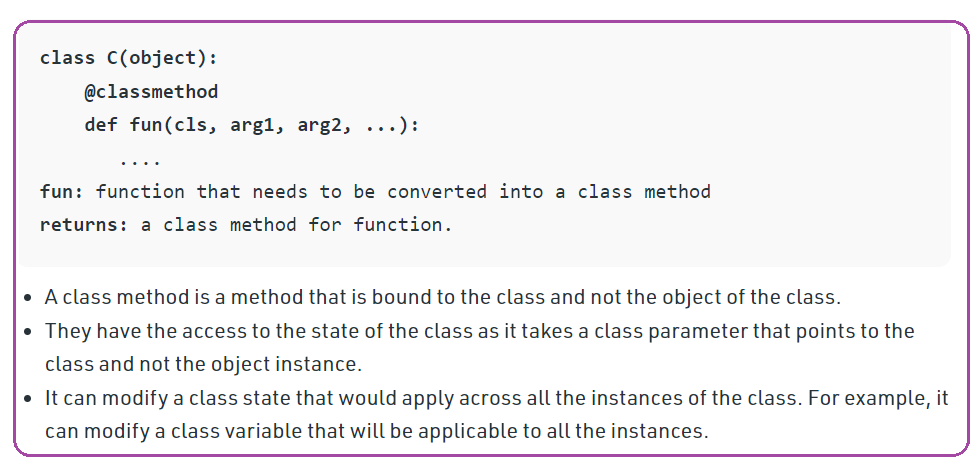
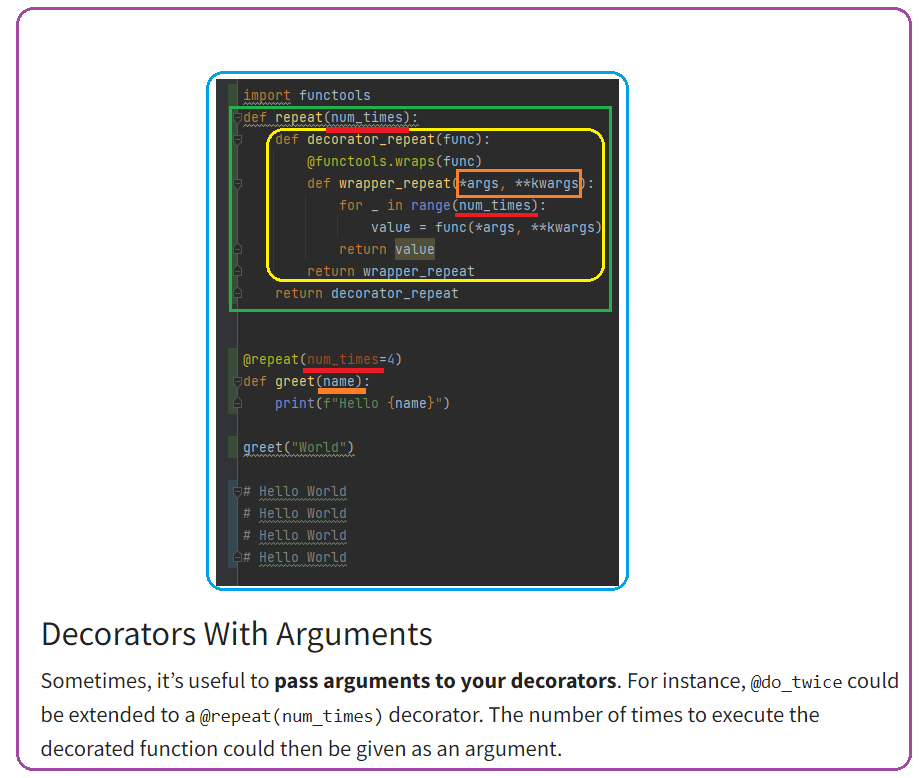
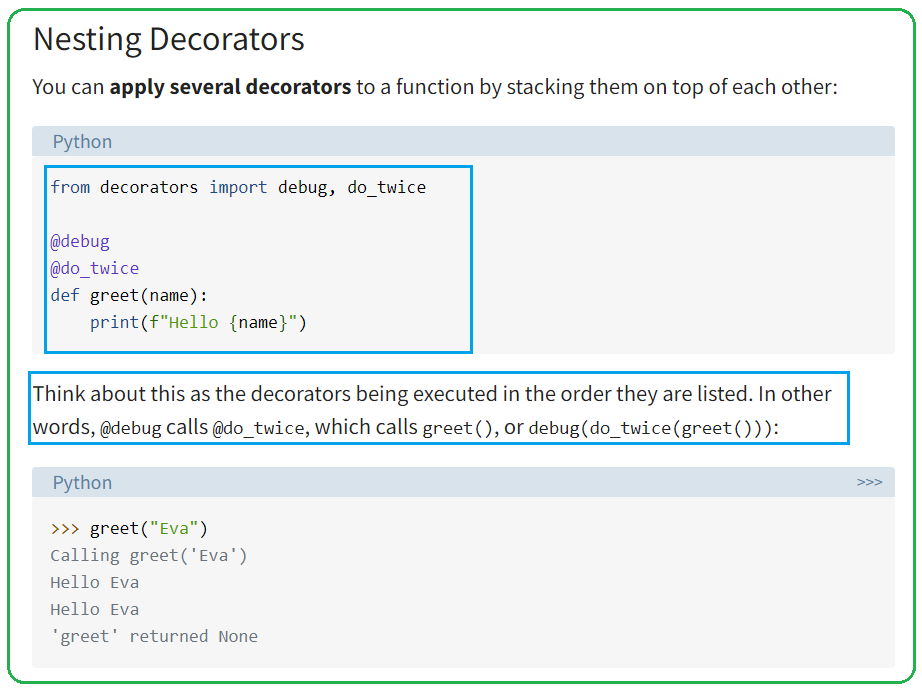
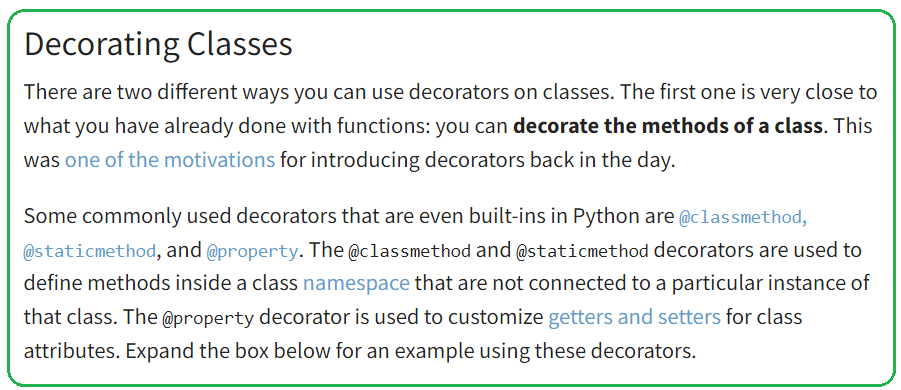
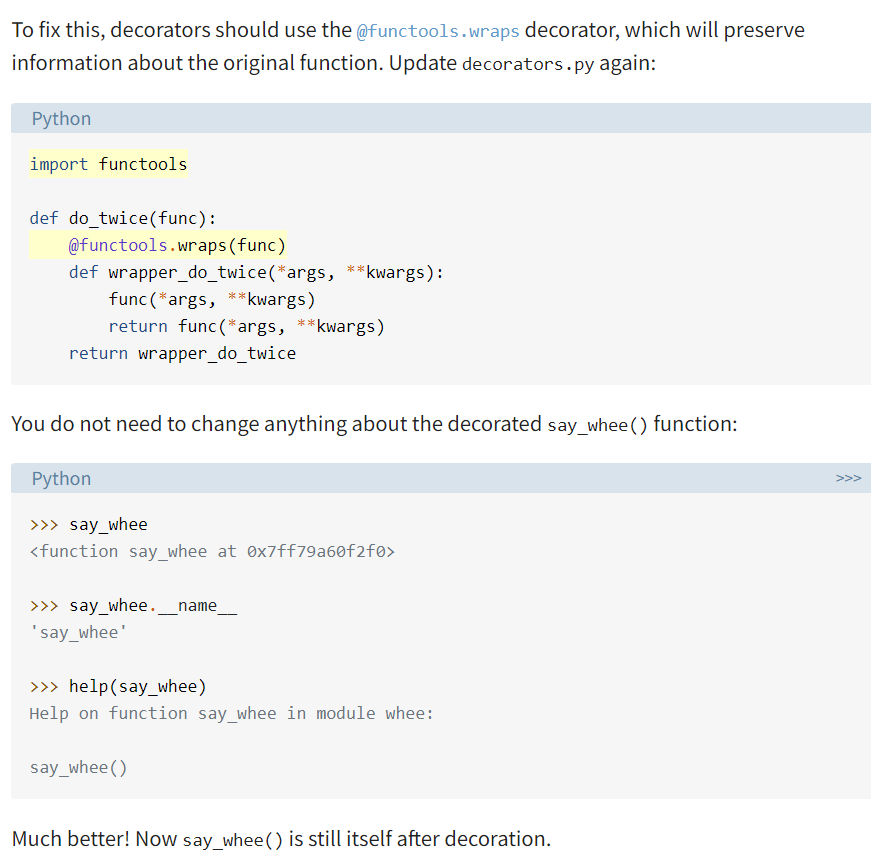
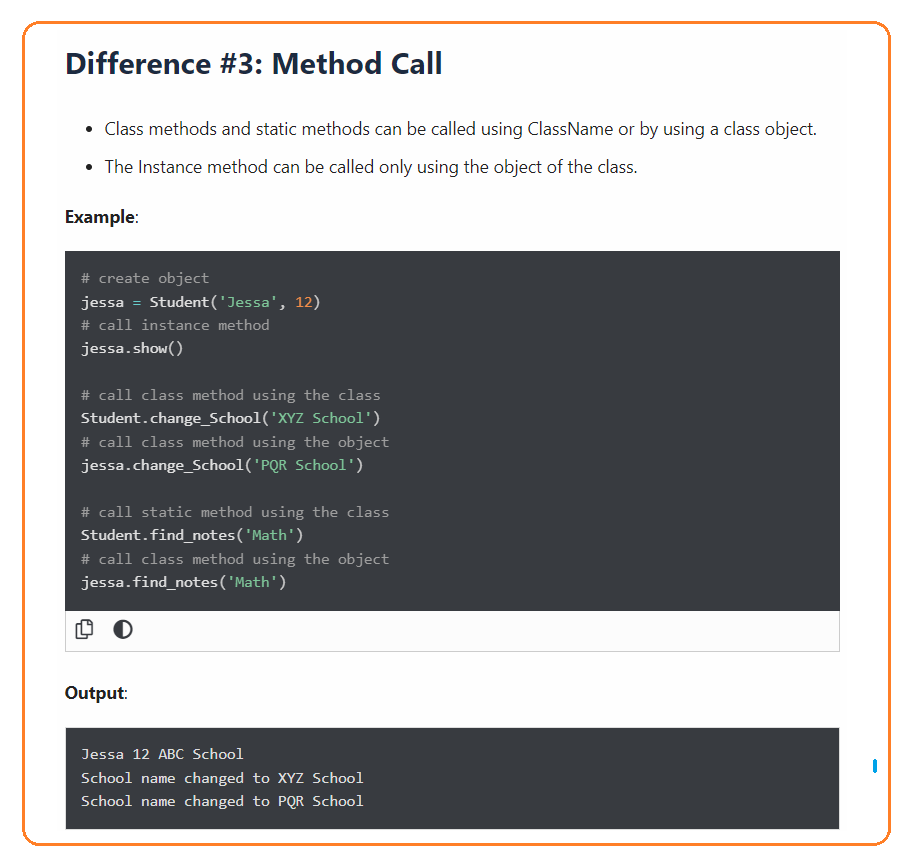
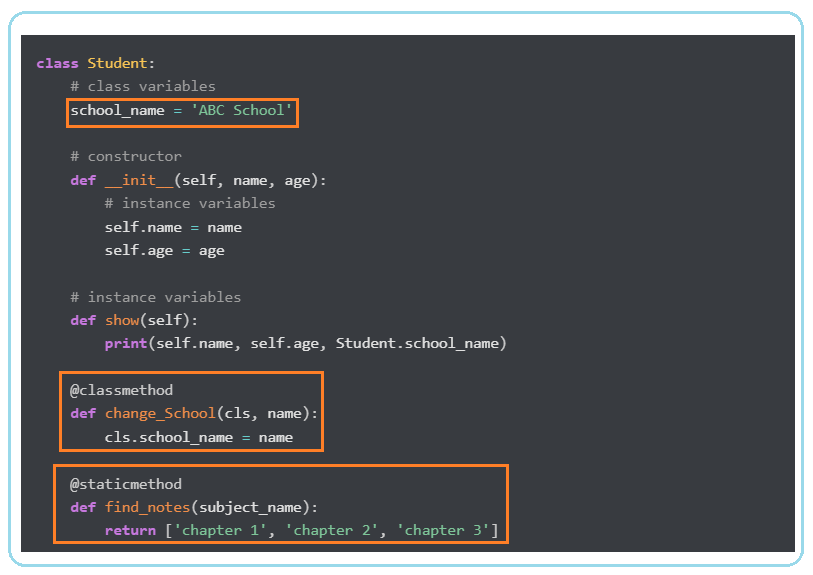
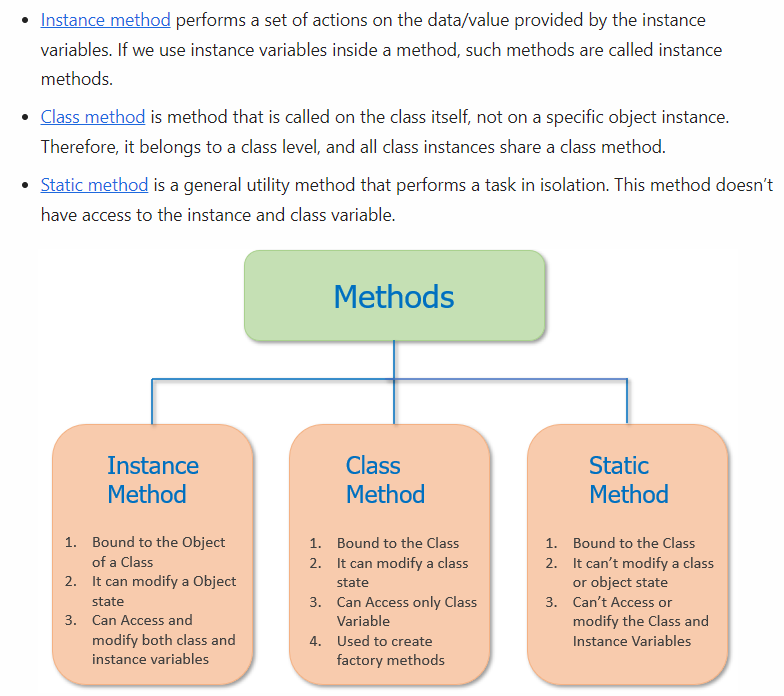
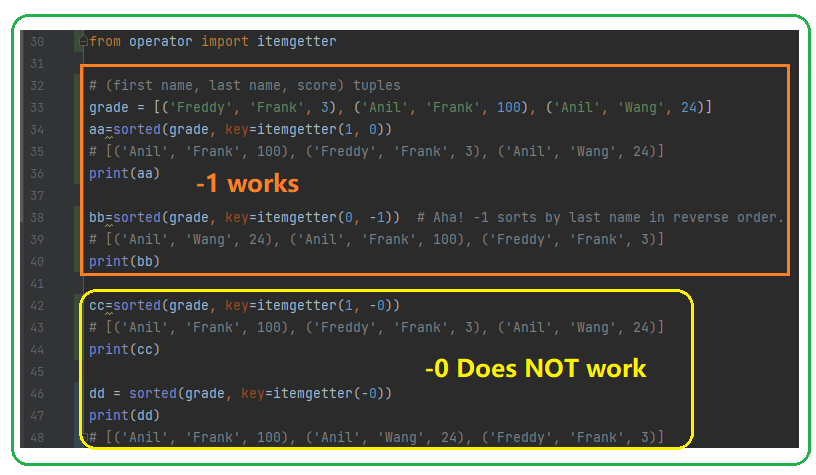
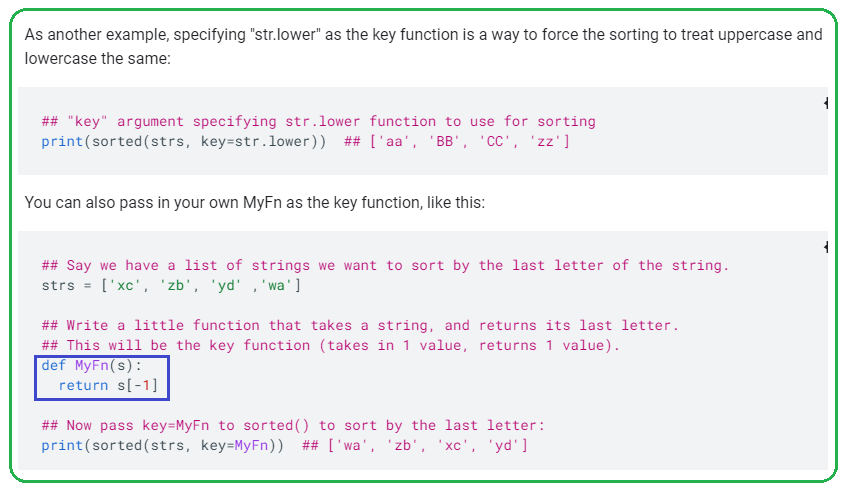
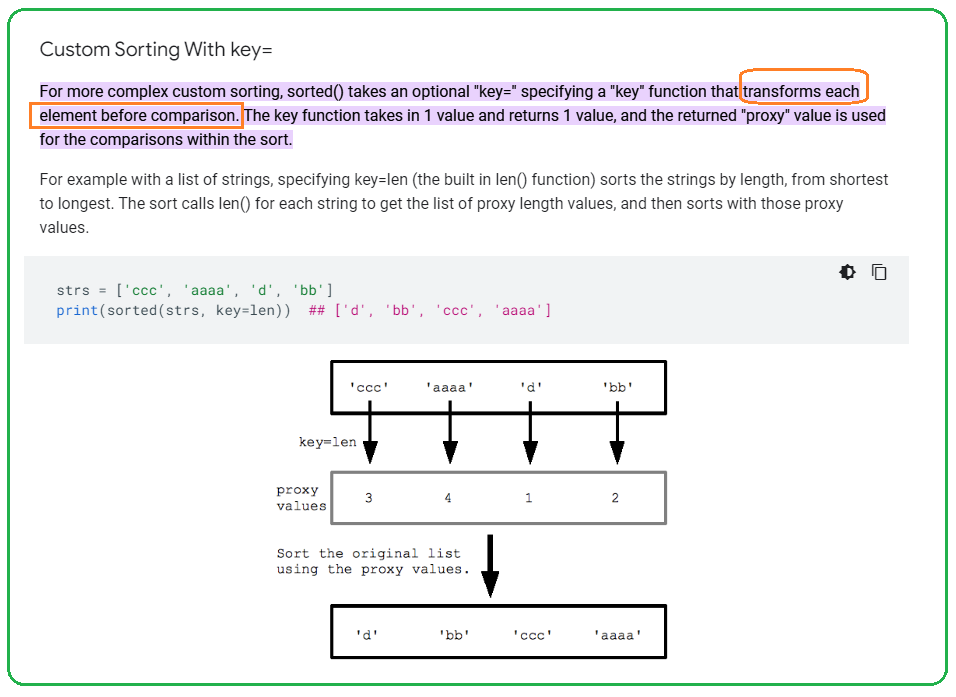
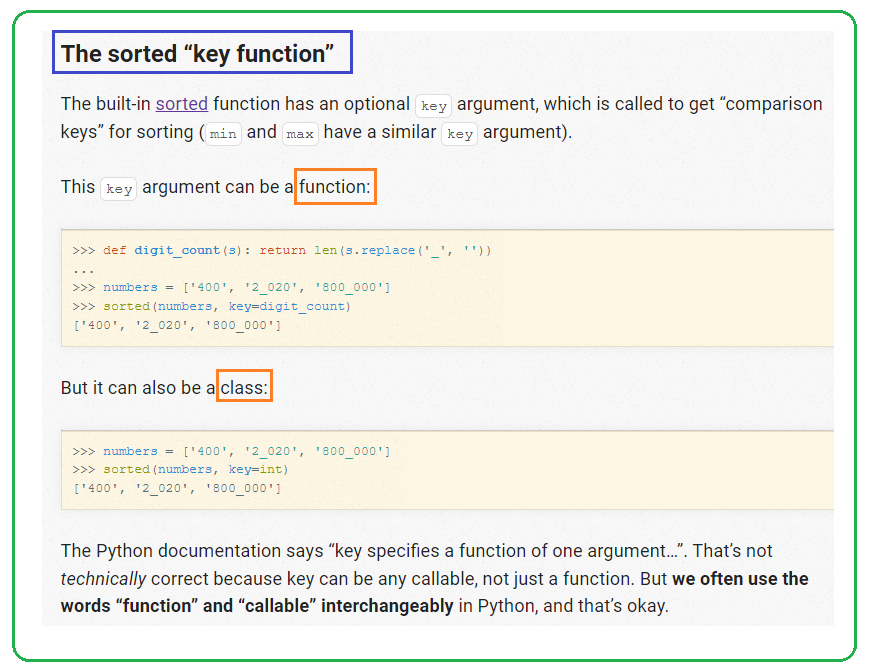
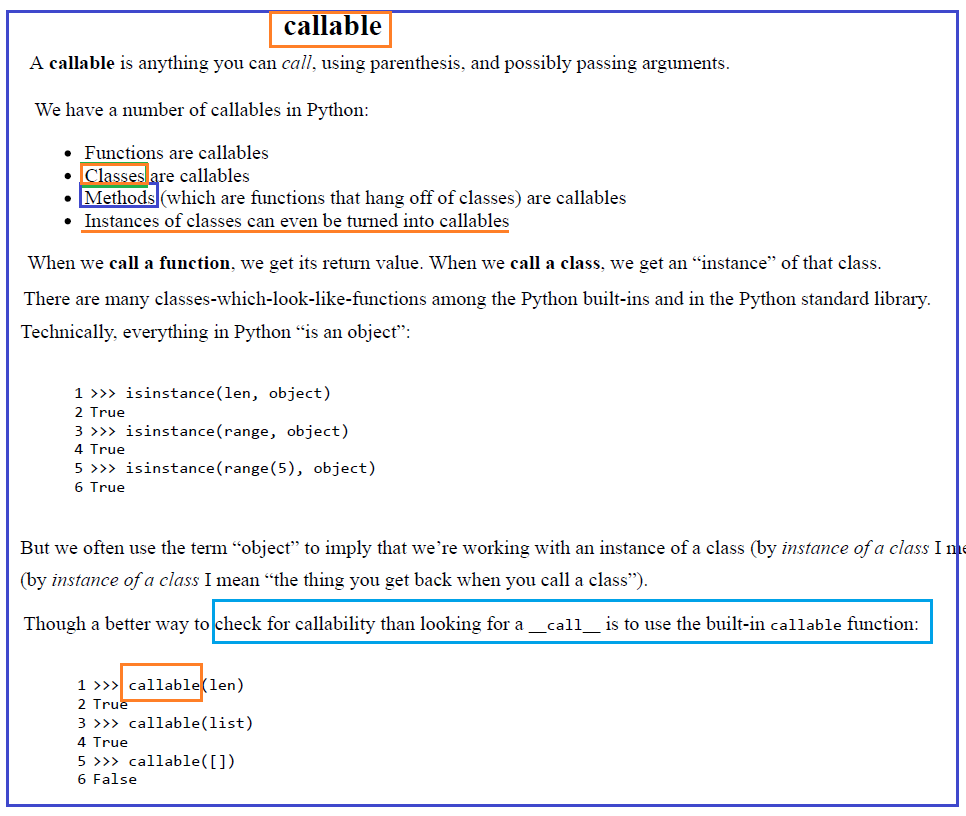
5/22/2021, 3/1/2022

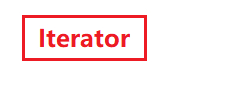
8/16/22, 9/29/22

<https://www.gairuo.com/p/python-object>

<https://medium.com/analytics-vidhya/sorted-function-using-key-parameter-in-python-7aa9b8cebfb6>







Python OOP

<https://www.python-course.eu/python3_properties.php>

[https://www.learnbyexample.org/**python-properties**/](https://www.learnbyexample.org/python-properties/)

<https://pythonguide.readthedocs.io/en/latest/python/property.html>

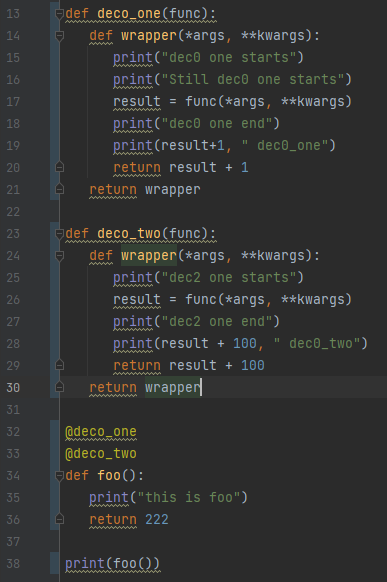
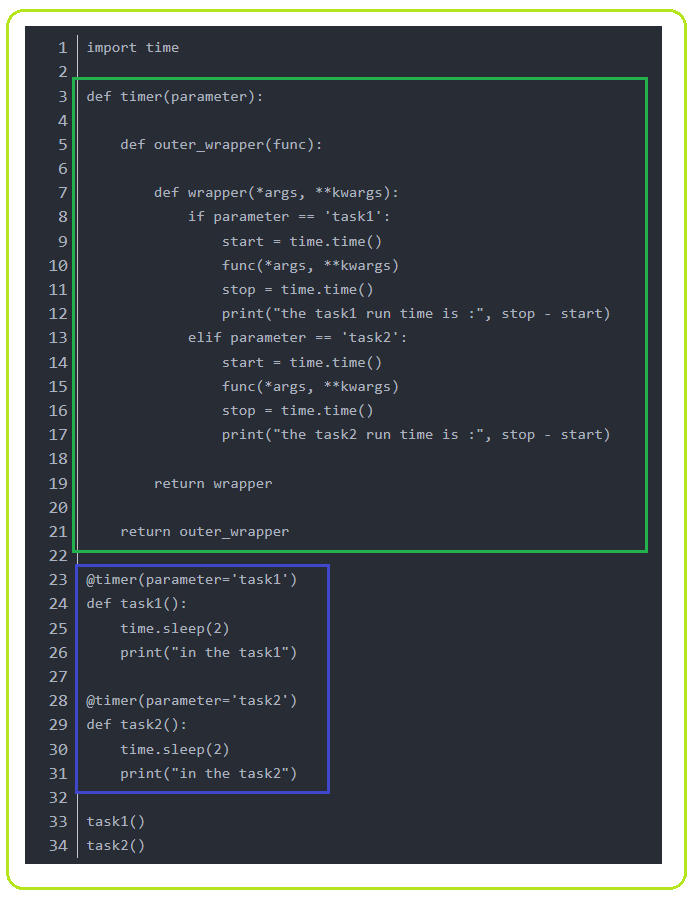
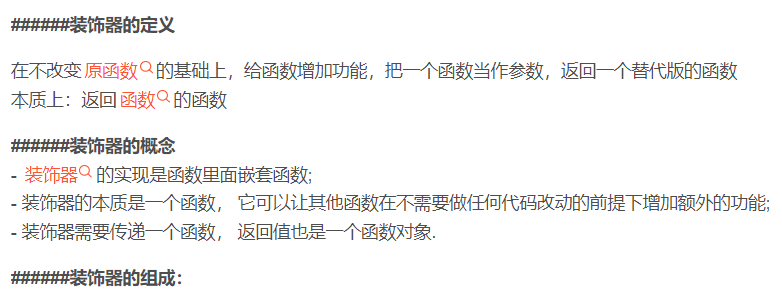
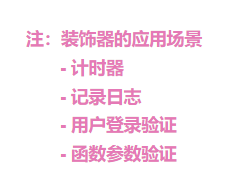
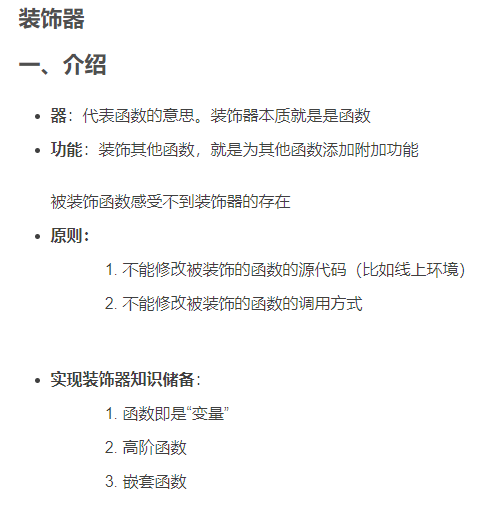
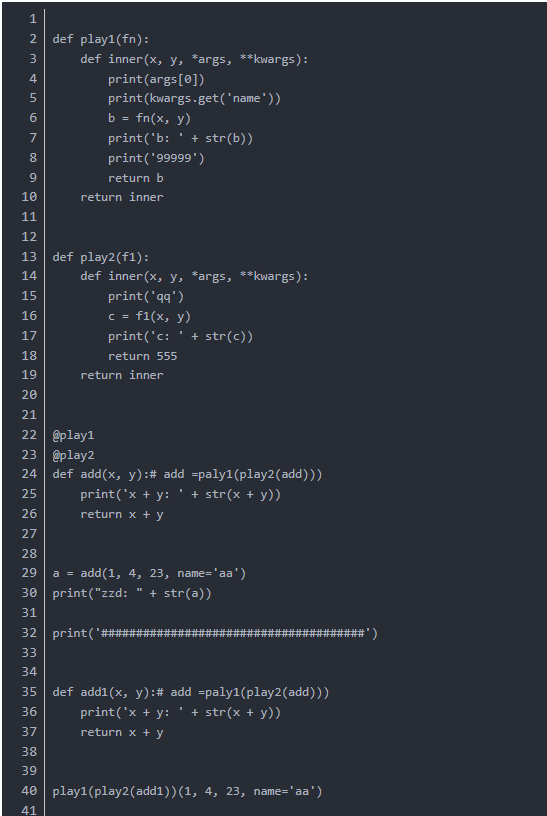
1. **python decorator**

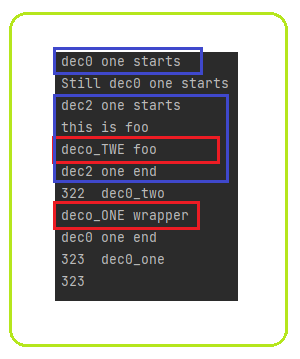
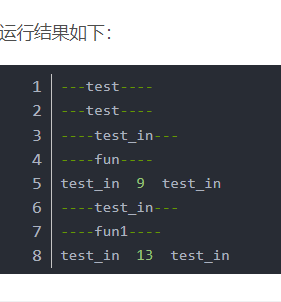
* A python @property decorator lets a method to be accessed as an attribute instead of as a method with a '()'.  
    
  To understand this, let’s create a Person class that contains the first, last and fullname of the person as attributes and has an email() method that provides the person’s email.  
    
  class Person():  
    
   def \_\_init\_\_(self, firstname, lastname):  
   self.first = firstname  
   self.last = lastname  
   self.fullname = self.first + ' '+ self.last  
    
   def email(self):  
   return '{}.{}@email.com'.format(self.first, self.last)  
  Let’s create an instance of the Person ‘selva prabhakaran’ and print the attributes.  
    
  # Create a Person object  
  person = Person('selva', 'prabhakaran')  
  print(person.first) #> selva  
  print(person.last) #> prabhakaran  
  print(person.fullname) #> selva prabhakaran  
  print(person.email()) #> selva.prabhakaran@email.com  
  3. When to use @property?  
  So far so good.  
    
  Now, somehow you decide to change the last name of the person.  
    
  Here is a fun fact about python classes: If you change the value of an attribute inside a class, **the other attributes that are derived from the attribute** you just changed **don’t automatically update**.  
    
  For example: By changing the **self.last name** you might expect the **self.full** attribute, which is derived from self.last to update. But **unexpectedly it doesn’t**. This can provide potentially misleading information about the person.  
    
  However, notice the email() works as intended, even though it is derived from self.last.  
    
  # Changing the `last` name does not change `self.full` name, but email() works  
  person.last = 'prasanna'  
  print(person.last) #> prasanna  
  print(person.fullname) #> selva prabhakaran  
  print(person.email()) #> selva.prasanna@email.com  
  So, a probable solution would be to convert the **self.fullname attribute** to a **fullname() method**, so it will provide correct value like the email() method did. Let’s do it.  
    
  # Converting fullname to a method provides the right fullname  
  # But it breaks old code that used the fullname attribute without the `()`  
  class Person():  
    
   def \_\_init\_\_(self, firstname, lastname):  
   self.first = firstname  
   self.last = lastname  
    
   def fullname(self):  
   return self.first + ' '+ self.last  
    
   def email(self):  
   return '{}.{}@email.com'.format(self.first, self.last)  
    
  person = Person('selva', 'prabhakaran')  
  print(person.fullname()) #> selva prabhakaran  
    
  # change last name to Prasanna  
  person.last = 'prasanna'  
    
  print(person.fullname()) #> selva prasanna  
  Now the convert to method solution works.  
    
  But there is a problem.  
    
  Since we are using **person.fullname() method with a '()' instead of person.fullname as attribute**, it will break whatever code that used the self.fullname attribute.  
  If you are building a product/tool, the chances are, other developers and users of your module used it at some point and all their code will break as well.  
    
  So a better solution (without breaking your user’s code) is to **convert the method as a property by adding a @property decorator** before the method’s definition.  
  # By doing this, the fullname() method can be accessed as an attribute instead of as a method with '()'. See example below.  
    
  # Adding @property provides the right fullname and does not break code!  
  class Person():  
    
   def \_\_init\_\_(self, firstname, lastname):  
   self.first = firstname  
   self.last = lastname  
    
   **@property  
   def fullname(self):  
   return self.first + ' '+ self.last**  
    
   def email(self):  
   return '{}.{}@email.com'.format(self.first, self.last)  
    
  # Init a Person  
  person = Person('selva', 'prabhakaran')  
  print(person.fullname) #> selva prabhakaran  
    
  # Change last name to Prasanna  
  person.last = 'prasanna'  
    
  # Print fullname  
  print(**person.fullname**) # selva prasanna  
  4. The setter method – When to use it and How to write one?  
  Now you are able to **access the fullname** like an attribute.  
    
  However there is one final problem.  
    
  Your users are going to want to change the fullname property at some point. And by setting it, they expect it will change the values of the first and last names from which fullname was derived in the first place.  
    
  But unfortunately, trying to set the value of fullname throws an AttributeError.  
    
  person.fullname = 'raja rajan'  
    
  #> ---------------------------------------------------------------------------  
  #> AttributeError Traceback (most recent call last)  
  #> <ipython-input-36-67cde7461cfc> in <module>  
  #> ----> 1 person.fullname = 'raja rajan'  
    
  #> AttributeError: can't set attribute  
  How to tackle this?  
    
  We define an equivalent setter method that will be called everytime a user sets a value to this property.  
    
  Inside this setter method, you can modify the values of variables that should be changed when the value of fullname is set/changed.  
    
  However, there are a couple of conventions you need to follow when defining a setter method:  
    
  The setter method should have the same name as the equivalent method that @property decorates.  
  It accepts as argument the value that user sets to the property.  
  Finally you need to add a @{methodname}.setter decorator just before the method definition.  
    
  Once you add the @{methodname}.setter decorator to it, this method will be called everytime the property (fullname in this case) is set or changed. See below.  
    
  class Person():  
    
   def \_\_init\_\_(self, firstname, lastname):  
   self.first = firstname  
   self.last = lastname  
    
   **@property  
   def fullname(self):  
   return self.first + ' '+ self.last**  
    
   **@fullname.setter  
   def fullname(self, name):  
   firstname, lastname = name.split()  
   self.first = firstname  
   self.last = lastname**  
    
   def email(self):  
   return '{}.{}@email.com'.format(self.first, self.last)  
    
  # Init a Person  
  person = Person('selva', 'prabhakaran')  
  print(person.fullname) #> selva prabhakaran  
  print(person.first) #> selva  
  print(person.last) #> prabhakaran  
    
  # Setting fullname calls the setter method and updates person.first and person.last  
  person.fullname = 'velu pillai'  
    
  # Print the changed values of `first` and `last`  
  print(person.fullname) #> velu pillai  
  print(person.first) #> pillai  
  print(person.last) #> pillai  
  There you go. We set a new value to person.fullname, the person.first and person.last updated as well. Our Person class will now automatically update the derived attributes (property) when one of the base attribute changes and vice versa.  
    
    
  5. The deleter method  
  Similar to the setter, the deleter’s method defines what happens when a property is deleted.  
    
  You can create the deleter method by defining a method of the same name and adding a @{methodname}.deleter decorator. See the implementation below.  
    
  class Person():  
    
   def \_\_init\_\_(self, firstname, lastname):  
   self.first = firstname  
   self.last = lastname  
    
   **@property  
   def fullname(self):  
   return self.first + ' '+ self.last**  
    
   **@fullname.setter  
   def fullname(self, name):  
   firstname, lastname = name.split()  
   self.first = firstname  
   self.last = lastname**  
    
   **@fullname.deleter  
   def fullname(self):  
   self.first = None  
   self.last = None**  
    
   def email(self):  
   return '{}.{}@email.com'.format(self.first, self.last)  
    
  # Init a Person  
  person = Person('selva', 'prabhakaran')  
  print(person.fullname) #> selva prabhakaran  
    
  # Deleting fullname calls the deleter method, which erases self.first and self.last  
  del person.fullname  
    
  # Print the changed values of `first` and `last`  
  print(person.first) #> None  
  print(person.last) #> None  
  In above case, the person.first and person.last attribute return None, once the fullname is deleted.  
    
  6. Conclusion  
  So, to summarize:  
    
  When to use @property decorator?  
  When an attribute is derived from other attributes in the class, so the derived attribute will update whenever the source attributes is changed.  
  How to make a @property?  
  Make an attribute as property by defining it as a function and add the @property decorator before the fn definition.  
  When to define a setter method for the property?  
  Typically, if you want to update the source attributes whenever the property is set. It lets you define any other changes as well

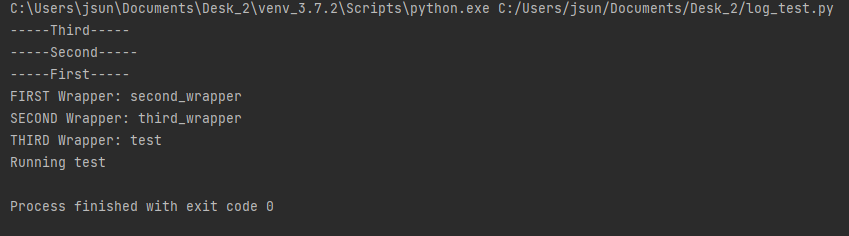
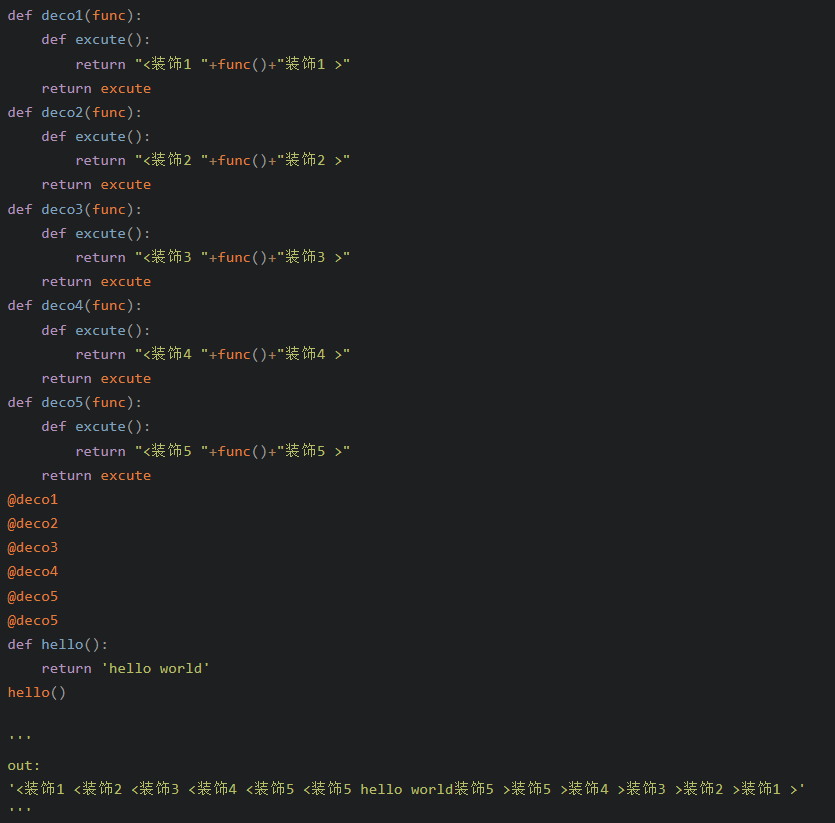
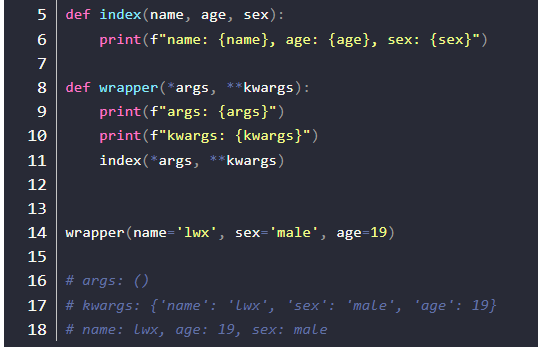
1. **TBD**

**** **** 

* [**https://blog.csdn.net/xiangxianghehe/article/details/77170585?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522164594579016781685378602%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request\_id=164594579016781685378602&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~top\_positive~default-1-77170585.pc\_search\_result\_positive&utm\_term=python%E8%A3%85%E9%A5%B0%E5%99%A8&spm=1018.2226.3001.4187**](https://blog.csdn.net/xiangxianghehe/article/details/77170585?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522164594579016781685378602%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=164594579016781685378602&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-77170585.pc_search_result_positive&utm_term=python%E8%A3%85%E9%A5%B0%E5%99%A8&spm=1018.2226.3001.4187)
* [**https://blog.csdn.net/fgf00/article/details/52319711?utm\_medium=distribute.pc\_relevant.none-task-blog-2~default~baidujs\_utm\_term~default-0.queryctrv2&spm=1001.2101.3001.4242.1&utm\_relevant\_index=3**](https://blog.csdn.net/fgf00/article/details/52319711?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_utm_term~default-0.queryctrv2&spm=1001.2101.3001.4242.1&utm_relevant_index=3)
* [**https://blog.csdn.net/five3/article/details/83447467?utm\_medium=distribute.pc\_relevant.none-task-blog-2~default~baidujs\_utm\_term~default-4.que**  **relevant\_index=7**](https://blog.csdn.net/five3/article/details/83447467?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_utm_term~default-4.que%20%20relevant_index=7)
* [**https://blog.csdn.net/duyun0/article/details/118087073?spm=1001.2101.3001.6650.2&utm\_medium=distribute.pc\_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7EHighlightScore-2.queryctrv2&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7EHighlightScore-2.queryctrv2&utm\_relevant\_index=5**](https://blog.csdn.net/duyun0/article/details/118087073?spm=1001.2101.3001.6650.2&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7EHighlightScore-2.queryctrv2&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7EHighlightScore-2.queryctrv2&utm_relevant_index=5)

**** ****         def deco\_one(func):  
 def wrapper(\*args, \*\*kwargs):  
 print("dec0 one starts")  
 print("Still dec0 one starts")  
 result = func(\*args, \*\*kwargs)  
 print(f"deco\_ONE {func.\_\_name\_\_}")  
 print("dec0 one end")  
 print(result+1, " dec0\_one")  
 return result + 1  
 return wrapper  
  
def deco\_two(func):  
 def wrapper(\*args, \*\*kwargs):  
 print("dec2 one starts")  
 result = func(\*args, \*\*kwargs)  
 print(f"deco\_TWE {func.\_\_name\_\_}")  
 print("dec2 one end")  
 print(result + 100, " dec0\_two")  
 return result + 100  
 return wrapper  
  
@deco\_one  
@deco\_two  
def foo():  
 print("this is foo")  
 return 222  
  
print(foo())

**** ****     def first\_deco(func):  
 print("-----First-----")  
  
 def first\_wrapper(\*args, \*\*kwargs):  
 print(f"FIRST Wrapper: {func.\_\_name\_\_}")  
 return func(\*args, \*\*kwargs)  
 return first\_wrapper  
  
  
def second\_deco(func):  
 print("-----Second-----")  
  
 def second\_wrapper(\*args, \*\*kwargs):  
 print(f"SECOND Wrapper: {func.\_\_name\_\_}")  
 return func(\*args, \*\*kwargs)  
  
 return second\_wrapper  
  
def third\_deco(func):  
 print("-----Third-----")  
  
 def third\_wrapper(\*args, \*\*kwargs):  
 print(f"THIRD Wrapper: {func.\_\_name\_\_}")  
 return func(\*args, \*\*kwargs)  
  
 return third\_wrapper  
  
@first\_deco  
@second\_deco  
@third\_deco  
def test():  
 print("Running test")  
  
test()

**** **** 

**Properties in Python**

Some object-oriented languages such as Java and C# support private object attributes; which cannot be directly accessed from outside. Programmers often have to write getter and setter methods to access such private attributes.

**However in Python, all the attributes and methods are public, so it is useless to write getters or setters**.

**If you want to prevent direct access to an attribute, you should define it as a property**.

It is a simple way to customize access to an attribute.

Python programming provides us with a built-in @property decorator which makes usage of getter and setters much easier in Object-Oriented Programming.

# Basic method of setting and getting attributes in Python  
class Celsius:  
 def \_\_init\_\_(self, temperature=0):  
 self.temperature = temperature  
  
 def to\_fahrenheit(self):  
 return (self.temperature \* 1.8) + 32  
  
  
# Create a new object  
human = Celsius()  
  
# Set the temperature  
human.temperature = 37  
  
# Get the temperature attribute  
print(human.temperature)  
  
# Get the to\_fahrenheit method  
print(human.to\_fahrenheit())

Whenever we assign or retrieve any object attribute like temperature as shown above, Python searches it in the object's built-in \_\_dict\_\_ dictionary attribute.

>>> human.\_\_dict\_\_

{'temperature': 37}

Therefore, man.temperature internally becomes man.\_\_dict\_\_['temperature'].

# Using @property decorator  
class Celsius:  
 def \_\_init\_\_(self, temperature=0):  
 self.\_temperature = temperature  
  
 def to\_fahrenheit(self):  
 return (self.\_temperature \* 1.8) + 32  
  
 @property  
 def temperature(self):  
 print("Getting value...")  
 return self.\_temperature  
  
 @temperature.setter  
 def temperature(self, value):  
 print("Setting value...")  
 if value < -273.15:  
 raise ValueError("Temperature below -273 is not possible")  
 self.\_temperature = value  
  
  
# create an object  
human = Celsius(37)  
print(human.temperature)  
# Getting value...  
# 37  
human.temperature = 100  
# Setting value...



class Person(object):  
 def \_\_init\_\_(self, first\_name, last\_name):  
 self.first\_name = first\_name  
 self.last\_name = last\_name  
  
 @property  
 def full\_name(self):  
 return self.first\_name + ' ' + self.last\_name  
  
 @full\_name.setter  
 def full\_name(self, value):  
 first\_name, last\_name = value.split(' ')  
 self.first\_name = first\_name  
 self.last\_name = last\_name  
  
 @full\_name.deleter  
 def full\_name(self):  
 del self.first\_name  
 del self.last\_name  
This is Python's way of creating getters, setters, and deleters (or mutator methods) for a property in a class.

## When do you use **class method**?

**https://www.programiz.com/python-programming/methods/built-in/classmethod**

### 1. Factory methods

Factory methods are those methods that return a class object (like constructor) for different use cases.

It is similar to function overloading in C++. Since, Python doesn't have anything as such, class methods and static methods are used.

### Example 2: Create factory method using class method

from datetime import date  
  
# random Person  
class **Person**:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 @classmethod  
 def fromBirthYear(cls, name, birthYear):  
 return cls(name, date.today().year - birthYear)  
  
 def display(self):  
 print(self.name + "'s age is: " + str(self.age))  
  
person = Person('Adam', 19)  
person.display()  
  
person1 = **Person**.fromBirthYear('John', 1985)  
person1.display()

# Adam's age is: 19  
# John's age is: 31

Here, **we have two class instance creator**, a constructor and a fromBirthYear method.

The constructor takes normal parameters name and age. While, fromBirthYear takes class, name and birthYear, calculates the current age by subtracting it with the current year and **returns the class instance**.

The fromBirthYear method takes Person class (not Person object) as the first parameter **cls** and returns the constructor by calling cls(name, date.today().year - birthYear), which is equivalent to Person(name, date.today().year - birthYear)

Before the method, we see @classmethod. This is called a [decorator](https://www.programiz.com/python-programming/decorator) for converting fromBirthYear to a class method as classmethod().

class Pizza:

def \_\_init\_\_(self, ingredients):

self.ingredients = ingredients

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

return f'Pizza({self.ingredients!r})'

**@classmethod**

def margherita(cls):

return **cls(['mozzarella', 'tomatoes'])**

@classmethod

def prosciutto(cls):

return cls(['mozzarella', 'tomatoes', 'ham'])

Note how I’m using the cls argument in the margherita and prosciutto factory methods instead of calling the Pizza constructor directly.

This is a trick you can use to follow the [Don’t Repeat Yourself (DRY)](https://en.wikipedia.org/wiki/Don't_repeat_yourself) principle. If we decide to rename this class at some point we won’t have to remember updating the constructor name in all of the classmethod factory functions.

Now, what can we do with these factory methods? Let’s try them out:

>>>

>>> Pizza.margherita()

Pizza(['mozzarella', 'tomatoes'])

>>> Pizza.prosciutto()

Pizza(['mozzarella', 'tomatoes', 'ham'])

As you can see, we can use the factory functions to create new Pizza objects that are configured the way we want them. They all use the same \_\_init\_\_ constructor internally and simply provide a shortcut for remembering all of the various ingredients.

**Another way to look at this use of class methods is that they allow you to define alternative constructors for your classes**.

Python only allows one \_\_init\_\_ method per class. Using class methods it’s possible to add as many alternative constructors as necessary. This can make the interface for your classes self-documenting (to a certain degree) and simplify their usage.

class Student(object):  
  
 def \_\_init\_\_(self, first\_name, last\_name):  
 self.first\_name = first\_name  
 self.last\_name = last\_name  
  
scott = Student('Scott', 'Robinson')

class Student(object):  
 def \_\_init\_\_(self, first\_name, last\_name):  
 self.first\_name = first\_name  
 self.last\_name = last\_name  
  
 @classmethod  
 def from\_string(cls, name\_str):  
 first\_name, last\_name = map(str, name\_str.split(' '))  
 student = cls(first\_name, last\_name)  
 return student  
  
scott = Student.from\_string('Scott Robinson')

class Student(object):  
  
 @classmethod  
 def from\_string(cls, name\_str):  
 first\_name, last\_name = map(str, name\_str.split(' '))  
 student = cls(first\_name, last\_name)  
 return student  
  
 @classmethod  
 def from\_json(cls, json\_obj):  
 # parse json...  
 return student  
  
 @classmethod  
 def from\_pickle(cls, pickle\_file):  
 # load pickle file...  
 return student

class **Student**(object):  
   
 def \_\_init\_\_(self, first\_name, last\_name):  
 self.first\_name = first\_name  
 self.last\_name = last\_name  
  
 @staticmethod  
 def is\_full\_name(name\_str):  
 names = name\_str.split(' ')  
 return len(names) > 1  
  
**Student**.is\_full\_name('Scott Robinson') # True  
**Student**.is\_full\_name('Scott')

## **What is a static method?**

Static methods, much like [class methods](https://www.programiz.com/python-programming/methods/built-in/classmethod), are methods that are bound to a class rather than its object.

## When do you use static methods?

### 1. **Grouping utility function to a class**

Static methods have a limited use case because, like class methods or any other methods within a class, they cannot access the properties of the class itself.

However, when you need a utility function that doesn't access any properties of a class but makes sense that it belongs to the class, we use static functions.

class **Dates**:  
 def \_\_init\_\_(self, date):  
 self.date = date  
  
 def getDate(self):  
 return self.date  
  
 **@staticmethod**  
 def **toDashDate**(date):  
 return date.replace("/", "-")  
  
  
date = Dates("15-12-2016")  
dateFromDB = "15/12/2016"  
dateWithDash = **Dates**.**toDashDate**(dateFromDB)  
  
if (date.getDate() == dateWithDash):  
 print("Equal")  
else:  
 print("Unequal")

**Output**

Equal

Here, we have a Dates class that only works with dates with dashes. However, in our previous database, all dates were present in slashes.

In order to convert the slash-dates to dash-dates, we have created a utility function toDashDate within Dates.

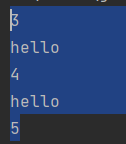
It is a static method because it doesn't need to access any properties of Dates itself and only requires the parameters.

We can also create toDashDate outside the class, but since it works only for dates, it's logical to keep it inside the Dates class.

* Global vs local



global a  
a = 3  
def Fuc():  
 global a  
 print(a)  
 a = a+1  
if \_\_name\_\_ == "\_\_main\_\_":  
 # global a  
 for i in range(2):  
 Fuc()  
 print('hello')  
 # global a  
 print(a)



NAME = ['ab','cd']  
def test1():  
 NAME = "erbi"  
 def test():  
 nonlocal NAME  
 NAME = "sb"  
 print(NAME)  
 test()  
 print(NAME)  
print(NAME)  
test1()  
print(NAME)  
  
# ['ab', 'cd']  
# sb  
# sb  
# ['ab', 'cd']

NAME = ['ab','cd']  
def test1():  
 NAME = "erbi"  
 def test():  
 nonlocal NAME  
 NAME = "sb"  
 print(NAME)  
 test()  
 print(NAME)  
print(NAME)  
test1()  
#   
# ['ab', 'cd']  
# sb  
# sb



# 软件测试面试中都会问到哪些关于Python的问题？

<https://blog.csdn.net/weixin_46635091/article/details/111209834?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-1.pc_relevant_paycolumn_v3&spm=1001.2101.3001.4242.2&utm_relevant_index=4>