Python - OOP & Decorator Tutorials

Object-Oriented Programming (OOP) Tutorials

Classes and Instances

In [1]:
```
class Employee:
    pass
```

In [2]:
```
emp_1 = Employee()
emp_2 = Employee()
print(emp_1)
print(emp_2)
```

```
<__main__.Employee object at 0x000001149D3979D0>
<__main__.Employee object at 0x000001149D3FF2D0>
```

In [3]:
```
emp_1.first = 'Emp1'
emp_1.last = 'User'
emp_1.email = 'Emp1.User@company.com'
emp_1.pay = 50000

emp_2.first = 'Emp2'
emp_2.last = 'User'
emp_2.email = 'Emp2.User@company.com'
emp_2.pay = 60000

print(emp_1.email)
print(emp_2.email)
```

```
Emp1.User@company.com
Emp2.User@company.com
```

In [4]:
```
class Employee:
    # Class init / Constructor
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first + '.' + last + '@company.com'

    def fullname(self):
        return f"{self.first} {self.last}"
```

In [5]:
```
emp_1 = Employee('Emp1', 'User', 50000)
emp_2 = Employee('Emp2', 'User', 60000)

print(emp_1.email)
print(emp_2.email)
```

```
Emp1.User@company.com
Emp2.User@company.com
```

In [6]:
```python
print(emp_1.fullname())
print(emp_2.fullname())
```

```
Emp1 User
Emp2 User
```

In [7]:
```python
print(Employee.fullname(emp_1))
print(Employee.fullname(emp_2))
```

```
Emp1 User
Emp2 User
```

Class Variables

In [8]:
```python
class Employee:

    num_of_emps = 0

    # Class init / Constructor
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first + '.' + last + '@company.com'

        Employee.num_of_emps += 1

    def fullname(self):
        return f"{self.first} {self.last}"
```

In [9]:
```python
print(Employee.num_of_emps)

emp_1 = Employee('Emp1', 'User', 50000)
emp_2 = Employee('Emp2', 'User', 60000)

print(Employee.num_of_emps)
```

```
0
2
```

In [10]:
```python
class Employee:

    num_of_emps = 0
    raise_amount = 1.04

    # Class init / Constructor
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first + '.' + last + '@company.com'

        Employee.num_of_emps += 1

    def fullname(self):
        return f"{self.first} {self.last}"
    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)
```

In [11]:
```
emp_1 = Employee('Emp1', 'User', 50000)
emp_2 = Employee('Emp2', 'User', 60000)

print(emp_1.pay)
emp_1.apply_raise()
print(emp_1.pay)

print(emp_2.pay)
emp_2.apply_raise()
print(emp_2.pay)
```

```
50000
52000
60000
62400
```

In [12]:
```
print(Employee.raise_amount)
print(emp_1.raise_amount)
print(emp_2.raise_amount)
```

```
1.04
1.04
1.04
```

In [13]:
```
print(emp_1.__dict__)
```

```
{'first': 'Emp1', 'last': 'User', 'pay': 52000, 'email': 'Emp1.User@company.co
m'}
```

In [14]:
```
print(Employee.__dict__)
```

```
{'__module__': '__main__', 'num_of_emps': 2, 'raise_amount': 1.04, '__init__':
<function Employee.__init__ at 0x000001149D4CE2A0>, 'fullname': <function Emplo
yee.fullname at 0x000001149D4CE340>, 'apply_raise': <function Employee.apply_ra
ise at 0x000001149D4CE3E0>, '__dict__': <attribute '__dict__' of 'Employee' obj
ects>, '__weakref__': <attribute '__weakref__' of 'Employee' objects>, '__doc_
_': None}
```

In [15]:
```
Employee.raise_amount = 1.05

print(Employee.raise_amount)
print(emp_1.raise_amount)
print(emp_2.raise_amount)
```

```
1.05
1.05
1.05
```

In [16]:
```
print(emp_1.__dict__)
```

```
{'first': 'Emp1', 'last': 'User', 'pay': 52000, 'email': 'Emp1.User@company.co
m'}
```

In [17]:
```
print(emp_2.__dict__)
```

```
{'first': 'Emp2', 'last': 'User', 'pay': 62400, 'email': 'Emp2.User@company.co
m'}
```

In [18]:
```
emp_1.raise_amount = 1.05
```

In [19]:
```
print(emp_1.__dict__)
```

```
{'first': 'Emp1', 'last': 'User', 'pay': 52000, 'email': 'Emp1.User@company.co
m', 'raise_amount': 1.05}
```

In [20]:
```
print(emp_2.__dict__)
```

```
{'first': 'Emp2', 'last': 'User', 'pay': 62400, 'email': 'Emp2.User@company.co
m'}
```

In [21]:
```
print(Employee.__dict__) # ???
```

```
{'__module__': '__main__', 'num_of_emps': 2, 'raise_amount': 1.05, '__init__':
<function Employee.__init__ at 0x000001149D4CE2A0>, 'fullname': <function Emplo
yee.fullname at 0x000001149D4CE340>, 'apply_raise': <function Employee.apply_ra
ise at 0x000001149D4CE3E0>, '__dict__': <attribute '__dict__' of 'Employee' obj
ects>, '__weakref__': <attribute '__weakref__' of 'Employee' objects>, '__doc_
_': None}
```

In [22]:
```
emp_1.raise_amount = 1.05

print(emp_1.pay)
emp_1.apply_raise()
print(emp_1.pay)

print(emp_2.pay)
emp_2.apply_raise()
print(emp_2.pay)
```

```
52000
54600
62400
65520
```

classmethods

In [23]:
```
class Employee:

    num_of_emps = 0
    raise_amt = 1.04

    # Class init / Constructor
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first + '.' + last + '@company.com'

        Employee.num_of_emps += 1

    def fullname(self):
        return f"{self.first} {self.last}"

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)

    @classmethod
    def set_raise_amt(cls, amount):
        cls.raise_amt = amount
```

In [24]:
```
print(Employee.raise_amt)

emp_1 = Employee('Emp1', 'User', 50000)
emp_2 = Employee('Emp2', 'User', 60000)

print(emp_1.raise_amt)
print(emp_2.raise_amt)
```

```
1.04
1.04
1.04
```

In [25]:
```
Employee.set_raise_amt(1.05)

print(Employee.raise_amt)
print(emp_1.raise_amt)
print(emp_2.raise_amt)
```

```
1.05
1.05
1.05
```

In [26]:
```
emp_str_1 = 'John-Doe-70000'
emp_str_2 = 'Steve-Smith-30000'

first, last, pay = emp_str_1.split('-')

new_emp_1 = Employee(first, last, pay)

print(new_emp_1.email)
print(new_emp_1.pay)
```

```
John.Doe@company.com
70000
```

In [27]:
```
class Employee:

    num_of_emps = 0
    raise_amt = 1.04

    # Class init / Constructor
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = int(pay)
        self.email = first + '.' + last + '@company.com'

        Employee.num_of_emps += 1

    def fullname(self):
        return f"{self.first} {self.last}"

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)

    @classmethod
    def set_raise_amt(cls, amount):
        cls.raise_amt = amount

    @classmethod
    def from_string(cls, emp_str):
        first, last, pay = emp_str.split('-')
        return cls(first, last, pay)
```

In [28]:
```
new_emp_2 = Employee.from_string(emp_str_2)

print(new_emp_2.email)
print(new_emp_2.pay)
```

```
Steve.Smith@company.com
30000
```

staticmethods

In [29]:
```
class Employee:

    num_of_emps = 0
    raise_amt = 1.04

    # Class init / Constructor
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = int(pay)
        self.email = first + '.' + last + '@company.com'

        Employee.num_of_emps += 1

    def fullname(self):
        return f"{self.first} {self.last}"

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)

    @classmethod
    def set_raise_amt(cls, amount):
        cls.raise_amt = amount

    @classmethod
    def from_string(cls, emp_str):
        first, last, pay = emp_str.split('-')
        return cls(first, last, pay)

    @staticmethod
    def is_workday(day):
        if day.weekday() == 5 or day.weekday() == 6:
            return False
        return True
```

In [30]:
```
import datetime
my_date = datetime.date(2016, 7, 10)
print(Employee.is_workday(my_date))
```

```
False
```

Inheritance - Creating Subclasses

In [31]:
```
# methond resolution chain

class Developer(Employee):
    pass
```

In [32]:
```
dev_str_1 = 'Jane-Doe-90000'
dev_1 = Developer.from_string(dev_str_1)

print(dev_1.email)
print(dev_1.pay)
```

```
Jane.Doe@company.com
90000
```

In [33]:
```
print(dev_1.pay)
dev_1.apply_raise()
print(dev_1.pay)
```

```
90000
93600
```

In [34]:
```
print(help(Developer))
```

```
Help on class Developer in module __main__:

class Developer(Employee)
 |  Developer(first, last, pay)
 |
 |  Method resolution order:
 |      Developer
 |      Employee
 |      builtins.object
 |
 |  Methods inherited from Employee:
 |
 |  __init__(self, first, last, pay)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  apply_raise(self)
 |
 |  fullname(self)
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from Employee:
 |
 |  from_string(emp_str) from builtins.type
 |
 |  set_raise_amt(amount) from builtins.type
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from Employee:
 |
 |  is_workday(day)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Employee:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from Employee:
 |
 |  num_of_emps = 1
 |
 |  raise_amt = 1.04

None
```

In [35]:

```python
class Developer(Employee):
    raise_amt = 1.1
```

In [36]:

```python
dev_str_1 = 'Jane-Doe-90000'
dev_1 = Developer.from_string(dev_str_1)

print(dev_1.email)
print(dev_1.pay)
```

```
Jane.Doe@company.com
90000
```

In [37]:
```
print(dev_1.pay)
dev_1.apply_raise()
print(dev_1.pay)
```

```
90000
99000
```

In [38]:
```
class Developer(Employee):
    raise_amt = 1.1

    def __init__(self, first, last, pay, prog_lang):
        super().__init__(first, last, pay)
        self.prog_lang = prog_lang
```

In [39]:
```
dev_2 = Developer('Corey', 'Schafer', 50000, 'Python')
```

In [40]:
```
print(dev_2.email)
print(dev_2.pay)
print(dev_2.prog_lang)
```

```
Corey.Schafer@company.com
50000
Python
```

In [41]:
```
class Manager(Employee):

    def __init__(self, first, last, pay, employees=None):
        super().__init__(first, last, pay)
        if employees is None:
            self.employees = []
        else:
            self.employees = employees

    def add_emp(self, emp):
        if emp not in self.employees:
            self.employees.append(emp)

    def remove_emp(self, emp):
        if emp in self.employees:
            self.employees.remove(emp)
    def print_emps(self):
        for emp in self.employees:
            print('--->', emp.fullname())
```

In [42]:
```
mgr_1 = Manager('Sue', 'Smith', 90000, [dev_1])

print(mgr_1.email)
mgr_1.print_emps()
```

```
Sue.Smith@company.com
---> Jane Doe
```

In [43]:
```
mgr_1.add_emp(dev_2)
mgr_1.print_emps()
```

```
---> Jane Doe
---> Corey Schafer
```

In [44]:
```
mgr_1.remove_emp(dev_1)
mgr_1.print_emps()
```

```
---> Corey Schafer
```

isinstance

In [45]:
```
print(isinstance(mgr_1, Manager))
```
```
True
```

In [46]:
```
print(isinstance(dev_1, Developer))
```
```
False
```

In [47]:
```
print(isinstance(dev_1, Employee))
```
```
True
```

In [48]:
```
print(isinstance(mgr_1, Employee))
```
```
True
```

In [49]:
```
print(isinstance(dev_1, Manager))
```
```
False
```

issubclass

In [50]:
```
print(issubclass(Developer, Employee))
```
```
True
```

In [51]:
```
print(issubclass(Developer, Manager))
```
```
False
```

Special (Magic/Dunder) Methods

> - `__repr__(self)` : the goal is to be umambiguous
> - `__str__(self)` : the goal is to readable

In [52]:
```
class Employee:

    num_of_emps = 0
    raise_amt = 1.04

    # Class init / Constructor
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = int(pay)
        self.email = first + '.' + last + '@company.com'

        Employee.num_of_emps += 1

    def fullname(self):
        return f"{self.first} {self.last}"

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)
    @classmethod
    def set_raise_amt(cls, amount):
        cls.raise_amt = amount

    @classmethod
    def from_string(cls, emp_str):
        first, last, pay = emp_str.split('-')
        return cls(first, last, pay)

    @staticmethod
    def is_workday(day):
        if day.weekday() == 5 or day.weekday() == 6:
            return False
        return True

    def __repr__(self):
        return "Employee('{}', '{}', {})".format(self.first, self.last, self.pay

    def __str__(self):
        return '{} - {}'.format(self.fullname(), self.email)

    def __add__(self, other):
        return self.pay + other.pay

    def __len__(self):
        return len(self.fullname())
```

In [53]:
```
emp_1 = Employee('Corey', 'Schafer', 50000)
emp_2 = Employee('Test', 'Employee', 60000)
```

In [54]:
```
print(emp_1)
print(emp_2)
```

```
Corey Schafer - Corey.Schafer@company.com
Test Employee - Test.Employee@company.com
```

In [55]:
```
print(emp_1.__repr__())
```

```
Employee('Corey', 'Schafer', 50000)
```

In [56]:
```
print(emp_1.__str__())
```

```
Corey Schafer - Corey.Schafer@company.com
```

In [57]:
```
print(emp_1.pay)
print(emp_2.pay)
```

```
50000
60000
```

In [58]:
```
print(emp_1 + emp_2)
```

```
110000
```

In [59]:
```
print(len(emp_1))
print(len(emp_2))
```

```
13
13
```

Property Decorators - Getters, Setters, and Deleters

In [60]:
```
class Employee:

    # Class init / Constructor
    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def email(self):
        return f"{self.first}.{self.last}@company.com"

    @property
    def fullname(self):
        return f"{self.first} {self.last}"

    @fullname.setter
    def fullname(self, name):
        first, last = name.split(' ')
        self.first = first
        self.last = last

    @fullname.deleter
    def fullname(self):
        print('Delete Name!')
        self.first = None
        self.last = None
```

In [61]:
```
emp_1 = Employee('John', 'Smith')
print(emp_1)
print(emp_1.first)
```

```
<__main__.Employee object at 0x000001149D50AC90>
John
```

In [62]:
```
emp_1.fullname = 'Corey Schafer'
print(emp_1)
print(emp_1.email)
print(emp_1.fullname)
```

```
<__main__.Employee object at 0x000001149D50AC90>
Corey.Schafer@company.com
Corey Schafer
```

In [63]:
```
del emp_1.fullname
```

```
Delete Name!
```

First-Class Functions

> In computer science, a programming language is said to have first-class functions if it treats functions as first-class citizens.

> This means the language supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures.

In [64]:
```
def square(x):
    return x * x

print(square)
```
```
<function square at 0x000001149D4A1D00>
```

In [65]:
```
f = square(5)
print(f)
```
```
25
```

In [66]:
```
f = square
print(f)
```
```
<function square at 0x000001149D4A1D00>
```

In [67]:
```
print(f(5))
```
```
25
```

> ## map
>
> `map function` takes a function and array as its arguments and it runs each value of the array through the provided function and then returns a new array of those results

In [68]:
```
def square(x):
    return x * x

def my_map(func, arg_list):
    result = []
    for i in arg_list:
        result.append(func(i)) # append the outcomes of funct
    return result

square = my_map(square, [1, 2, 3, 4, 5])

print(square)
```
```
[1, 4, 9, 16, 25]
```

In [69]:
```
def cube(x):
    return x * x * x

cube = my_map(cube, [1, 2, 3, 4, 5])

print(cube)
```
```
[1, 8, 27, 64, 125]
```

Closures

> - retutn function

In [70]:
```
def outer_function():
    message = 'Hi'
    def inner_function():
        print(message)
    return inner_function()

outer_function()
```

```
Hi
```

In [71]:
```
def outer_function():
    message = 'Hi'
    def inner_function():
        print(message)
    return inner_function

my_func = outer_function() # Return a function without executing

my_func()
my_func()
my_func()
```

```
Hi
Hi
Hi
```

In [72]:
```
def outer_function(msg):
    message = msg
    def inner_function():
        print(message)
    return inner_function

hi_func = outer_function('hi') # Return a function without executing
bye_func = outer_function('bye') # Return a function without executing

hi_func()
bye_func()
```

```
hi
bye
```

In [73]:
```
def decorator_function(message):
    def wrapper_function():
        print(message)
    return wrapper_function

hi_func = decorator_function('hi') # Return a function without executing
bye_func = decorator_function('bye') # Return a function without executing

hi_func()
bye_func()
```

```
hi
bye
```

In [74]:
```
def html_tag(tag):
    def wrap_text(msg):
        #print('<{0}>{1}</{0}>'.format(tag, msg))
        print(f"<{tag}>{msg}</{tag}>")
    return wrap_text


print_h1 = html_tag('h1') # Return a function without executing
print_h1('Test Headline!')
print_h1('Another Headline!')

print_p = html_tag('p')
print_p('Test Paragraph!')
```

```
<h1>Test Headline!</h1>
<h1>Another Headline!</h1>
<p>Test Paragraph!</p>
```

Decorator Tutorials

Dynamically Alter The Functionality Of Your Functions

In [75]:
```
def decorator_function(original_function):
    def wrapper_function():
        #print('wrapper exected this before {}'.format(original_function.__name_
        print(f'wrapper exected this before {original_function.__name__}')
        return original_function()
    return wrapper_function

def display():
    print('display function ran')

decorated_display = decorator_function(display) # Return a function without exec

decorated_display()
```

```
wrapper exected this before display
display function ran
```

In [76]:
```
def decorator_function(original_function):
    def wrapper_function():
        #print('wrapper exected this before {}'.format(original_function.__name_
        print(f'wrapper exected this before {original_function.__name__}')
        return original_function()
    return wrapper_function


@decorator_function
def display():
    print('display function ran - @decorator_function')

display()
```

```
wrapper exected this before display
display function ran - @decorator_function
```

In [77]:
```
def decorator_function(original_function):
    def wrapper_function(*args, **kwargs):
        #print('wrapper exected this before {}'.format(original_function.__name_
        print(f'wrapper exected this before {original_function.__name__}')
        return original_function(*args, **kwargs)
    return wrapper_function


@decorator_function
def display():
    print('display function ran - @decorator_function')

@decorator_function
def display_info(name, age):
    print(f'display_info ran - @decorator_function with arguments ({name}, {age}

display()
print('\n')
display_info('John', 25)
```

```
wrapper exected this before display
display function ran - @decorator_function


wrapper exected this before display_info
display_info ran - @decorator_function with arguments (John, 25)
```

In [78]:

```
class decorator_class(object):
    def __init__(self, original_function):
        self.original_function = original_function

    def __call__(self, *args, **kwargs):
        print(f'Call method exected this before {self.original_function.__name__
        return self.original_function(*args, **kwargs)

@decorator_class
def display():
    print('display function ran - @decorator_function')

@decorator_class
def display_info(name, age):
    print(f'display_info ran - @decorator_function with arguments ({name}, {age}

display()
print('\n')
display_info('John', 25)
```

```
Call method exected this before display
display function ran - @decorator_function


Call method exected this before display_info
display_info ran - @decorator_function with arguments (John, 25)
```

In [79]:

```
def my_logger(orig_func):
    import logging
    logging.basicConfig(filename=f'{orig_func.__name__}.log', level=logging.INFO

    def wrapper(*args, **kwargs):
        logging.info(f"Ran with args: {args} and kwargs: {kwargs}")
        return orig_func(*args, **kwargs)

    return wrapper
```

```
@my_logger
def display_info(name, age):
    print(f'display_info ran - @decorator_function with arguments ({name}, {age}


display_info('John', 25)
```

```
display_info ran - @decorator_function with arguments (John, 25)
```

In [80]:

```
import time

def my_timer(orig_func):
    import time

    def wrapper(*args, **kwargs):
        t1 = time.time()
        result = orig_func(*args, **kwargs)
        t2 = time.time() - t1
        print(f"{orig_func.__name__} ran in: {t2} sec")
        return result

    return wrapper


@my_timer
def display_info(name, age):
    time.sleep(1)
    print(f'display_info ran @decorator_function with arguments ({name}, {age})'

display_info('John', 25)
```

```
display_info ran @decorator_function with arguments (John, 25)
display_info ran in: 1.000685691833496 sec
```

In [81]:

```
def my_logger(orig_func):
    import logging
    #logging.basicConfig(filename='{}.log'.format(orig_func.__name__), level=log
    logging.basicConfig(filename=f'{orig_func.__name__}.log', level=logging.INFO

    def wrapper(*args, **kwargs):
        logging.info(f"Ran with args: {args} and kwargs: {kwargs}")
        return orig_func(*args, **kwargs)

    return wrapper

def my_timer(orig_func):
    import time

    def wrapper(*args, **kwargs):
        t1 = time.time()
        result = orig_func(*args, **kwargs)
        t2 = time.time() - t1
        print(f"{orig_func.__name__} ran in: {t2} sec")
        return result

    return wrapper


@my_logger
@my_timer
```

```
def display_info(name, age):
    time.sleep(1)
    print(f'display_info ran @my_timer then @my_logger with arguments ({name}, {

display_info('John', 25)
```

```
display_info ran @my_timer then @my_logger with arguments (John, 25)
display_info ran in: 1.001542329788208 sec
```

In [82]:
```
def my_logger(orig_func):
    import logging
    #logging.basicConfig(filename='{}.log'.format(orig_func.__name__), level=log
    logging.basicConfig(filename=f'{orig_func.__name__}.log', level=logging.INFO

    def wrapper(*args, **kwargs):
        logging.info(f"Ran with args: {args} and kwargs: {kwargs}")
        return orig_func(*args, **kwargs)

    return wrapper

def my_timer(orig_func):
    import time

    def wrapper(*args, **kwargs):
        t1 = time.time()
        result = orig_func(*args, **kwargs)
        t2 = time.time() - t1
        print(f"{orig_func.__name__} ran in: {t2} sec")
        return result

    return wrapper



@my_timer
@my_logger
def display_info(name, age):
    time.sleep(1)
    print(f'display_info ran @my_logger then @my_timer with arguments ({name}, {

display_info('Eric', 55)
```

```
display_info ran @my_logger then @my_timer with arguments (Eric, 55)
wrapper ran in: 1.0023927688598633 sec
```

In [83]:
```
# fix wrap
from functools import wraps

def my_logger(orig_func):
    import logging
    #logging.basicConfig(filename='{}.log'.format(orig_func.__name__), level=log
    logging.basicConfig(filename=f'{orig_func.__name__}.log', level=logging.INFO

    @wraps(orig_func)
    def wrapper(*args, **kwargs):
        logging.info(f"Ran with args: {args} and kwargs: {kwargs}")
        return orig_func(*args, **kwargs)

    return wrapper

def my_timer(orig_func):
    import time

    @wraps(orig_func)
    def wrapper(*args, **kwargs):
        t1 = time.time()
        result = orig_func(*args, **kwargs)
        t2 = time.time() - t1
        print(f"{orig_func.__name__} ran in: {t2} sec")
        return result

    return wrapper



@my_timer
@my_logger
def display_info(name, age):
    time.sleep(1)
    print(f'display_info ran @my_logger then @my_timer with arguments ({name}, {

display_info('Tome', 20)
```

```
display_info ran @my_logger then @my_timer with arguments (Tome, 20)
display_info ran in: 1.003450632095337 sec
```

Decorators With Arguments

In [84]:
```
def decorator_function(original_function):
    def wrapper_function(*args, **kwargs):
        print('Executed Before', original_function.__name__)
        result = original_function(*args, **kwargs)
        print('Executed After', original_function.__name__, '\n')
        return result
    return wrapper_function


@decorator_function
def display_info(name, age):
    print('display_info ran with arguments ({}, {})'.format(name, age))


display_info('John', 25)
display_info('Travis', 30)
```

```
Executed Before display_info
display_info ran with arguments (John, 25)
Executed After display_info

Executed Before display_info
display_info ran with arguments (Travis, 30)
Executed After display_info
```

In [85]:
```
def prefix_decorator(prefix):
    def decorator_function(original_function):
        def wrapper_function(*args, **kwargs):
            print(prefix, 'Executed Before', original_function.__name__)
            result = original_function(*args, **kwargs)
            print(prefix, 'Executed After', original_function.__name__, '\n')
```

```
                return result
            return wrapper_function
        return decorator_function


@prefix_decorator('LOG:')
def display_info(name, age):
    print('display_info ran with arguments ({}, {})'.format(name, age))


display_info('John', 25)
display_info('Travis', 30)
```

```
LOG: Executed Before display_info
display_info ran with arguments (John, 25)
LOG: Executed After display_info

LOG: Executed Before display_info
display_info ran with arguments (Travis, 30)
LOG: Executed After display_info
```

Logging to Files, Setting Levels, and Formatting

> https://docs.python.org/3/library/logging.html

> - `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.

In [89]:

```
def add(x, y):
    """Add Function"""
    return x + y


def subtract(x, y):
    """Subtract Function"""
    return x - y


def multiply(x, y):
    """Multiply Function"""
    return x * y


def divide(x, y):
    """Divide Function"""
    return x / y
```

```python
num_1 = 20
num_2 = 10

add_result = add(num_1, num_2)

print('Add: {} + {} = {}'.format(num_1, num_2, add_result))

sub_result = subtract(num_1, num_2)
print('Sub: {} - {} = {}'.format(num_1, num_2, sub_result))

mul_result = multiply(num_1, num_2)
print('Mul: {} * {} = {}'.format(num_1, num_2, mul_result))

div_result = divide(num_1, num_2)
print('Div: {} / {} = {}'.format(num_1, num_2, div_result))
```

```
Add: 20 + 10 = 30
Sub: 20 - 10 = 10
Mul: 20 * 10 = 200
Div: 20 / 10 = 2.0
```

In [92]:
```python
import logging

logging.basicConfig(filename="test.log", level=logging.WARNING)

def add(x, y):
    """Add Function"""
    return x + y

def subtract(x, y):
    """Subtract Function"""
    return x - y

def multiply(x, y):
    """Multiply Function"""
    return x * y

def divide(x, y):
    """Divide Function"""
    return x / y


num_1 = 20
num_2 = 10

add_result = add(num_1, num_2)

logging.warning('Add: {} + {} = {}'.format(num_1, num_2, add_result))

sub_result = subtract(num_1, num_2)
logging.warning('Sub: {} - {} = {}'.format(num_1, num_2, sub_result))

mul_result = multiply(num_1, num_2)
logging.warning('Mul: {} * {} = {}'.format(num_1, num_2, mul_result))

div_result = divide(num_1, num_2)
logging.warning('Div: {} / {} = {}'.format(num_1, num_2, div_result))
```

In [100…
```
import logging

logging.basicConfig(
    filename="test.log",
    level=logging.WARNING,
    format='%(asctime)s:%(levelname)s:%(message)s'
)

def add(x, y):
    """Add Function"""
    return x + y

def subtract(x, y):
    """Subtract Function"""
    return x - y

def multiply(x, y):
    """Multiply Function"""
    return x * y

def divide(x, y):
    """Divide Function"""
    return x / y


num_1 = 9
num_2 = 3

add_result = add(num_1, num_2)

logging.warning('Add: {} + {} = {}'.format(num_1, num_2, add_result))

sub_result = subtract(num_1, num_2)
logging.warning('Sub: {} - {} = {}'.format(num_1, num_2, sub_result))

mul_result = multiply(num_1, num_2)
logging.warning('Mul: {} * {} = {}'.format(num_1, num_2, mul_result))

div_result = divide(num_1, num_2)
logging.warning('Div: {} / {} = {}'.format(num_1, num_2, div_result))
```

In [101…
```
import logging

logging.basicConfig(filename='employee.log',
                    level=logging.INFO,
                    format='%(levelname)s:%(message)s')


class Employee:
    """A sample Employee class"""

    def __init__(self, first, last):
        self.first = first
        self.last = last

        logging.info('Created Employee: {} - {}'.format(self.fullname, self.emai

    @property
    def email(self):
        return '{}.{}@email.com'.format(self.first, self.last)

    @property
    def fullname(self):
        return '{} {}'.format(self.first, self.last)


emp_1 = Employee('John', 'Smith')
emp_2 = Employee('Corey', 'Schafer')
emp_3 = Employee('Jane', 'Doe')
```

-- MEMO END --