WRITING PYTHONIC CODES

Filling your holes in **python.**

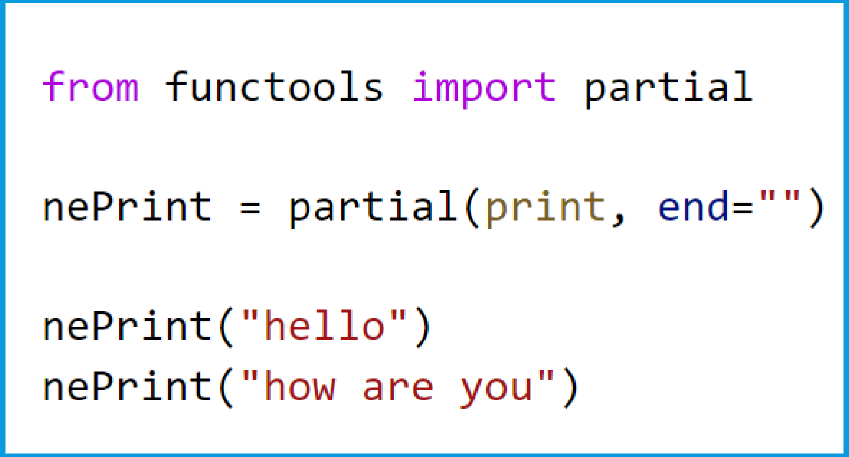
@regmicmahesh

~~FUNTOOLS~~ FUNCTOOLS

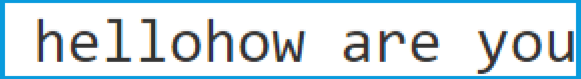
- The functools module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.
- This is also known as “currying”, a term named after FP pioneer Haskell Curry.
- Let’s start with what is functional programming.

PARTIAL FUNCTION

- Partial function are decorated function in which some arguments are already supplied.
- In other words, it modifies a function in such a way not every parameter is required in new function calls.



```python
from functools import partial

nePrint = partial(print, end="")

nePrint("hello")
nePrint("how are you")
```

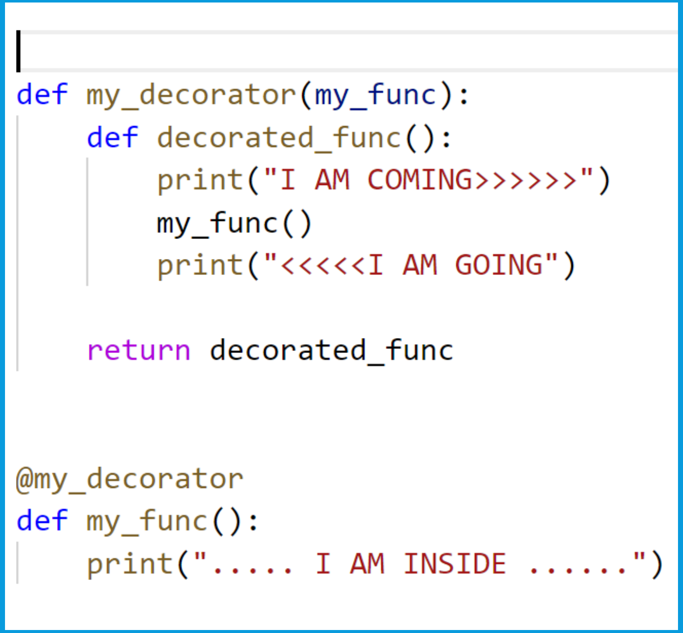


```
hellohow are you
```

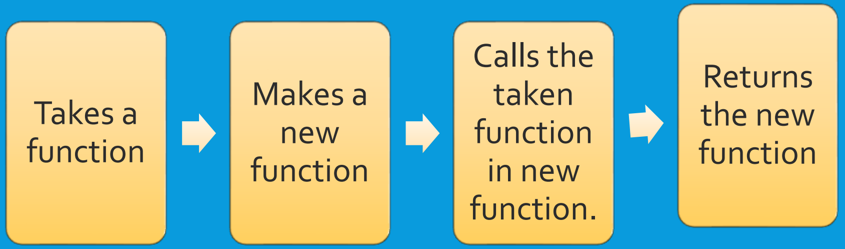
Keep in mind, it doesn't change docstrings, execution order or properties of the functions, it just **decorates** to use those kwargs everytime in function call.

DECORATES? WHAT?

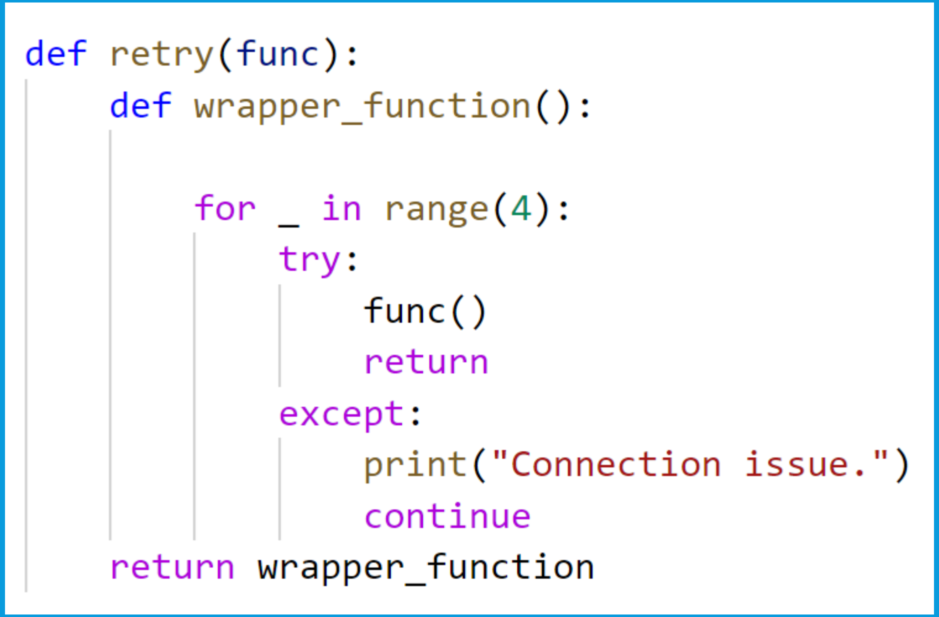
- When I decorate, I take the function and can modify what happens before the function call and what happens after the function call.
- I also may never call the function.
- Just a little but powerful feature enabled due to higher order functions.



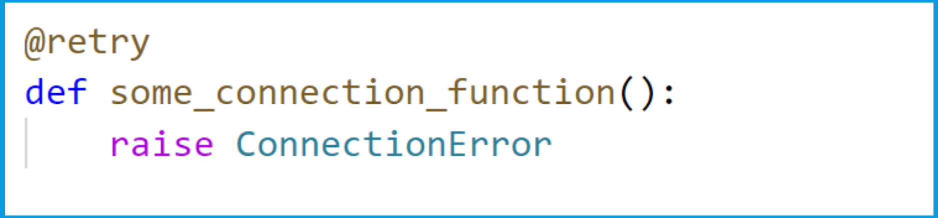
```
def my_decorator(my_func):
    def decorated_func():
        print("I AM COMING>>>>>>")
        my_func()
        print("<<<<<I AM GOING")

    return decorated_func


@my_decorator
def my_func():
    print("..... I AM INSIDE ......")
```



A LITTLE USEFUL EXAMPLE..



```python
def retry(func):
    def wrapper_function():

        for _ in range(4):
            try:
                func()
                return
            except:
                print("Connection issue.")
                continue
    return wrapper_function
```



```python
@retry
def some_connection_function():
    raise ConnectionError
```

CONCLUDING…

- You can research further about nesting decorators, repeating decorators and also stacking decorators.
- Decorators is a cool concept which lets you make a base function and use that function is a variety of ways.
- Think of the login, auth decorators in Django as an example.

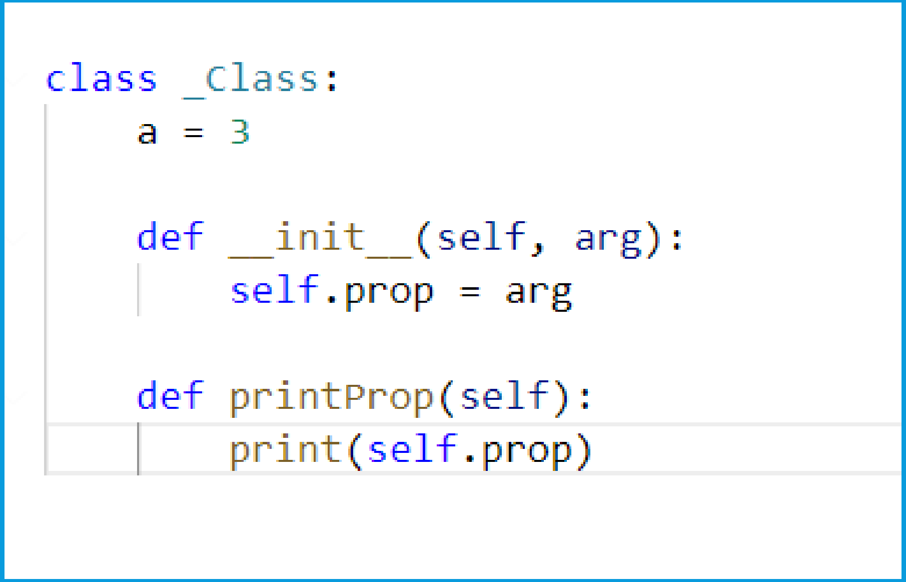
Just a little technical warning, classes decorators also exists. But generally decorators are used with functions.

CLASSES (WITH DATA)

JUST A SIMPLE CLASS.

- Just making sure we all have a fundamental understanding of class.

- __init__ is called dunder method (as it starts with double underscore) also known as initializer or constructor.



```
class _Class:
    a = 3

    def __init__(self, arg):
        self.prop = arg

    def printProp(self):
        print(self.prop)
```

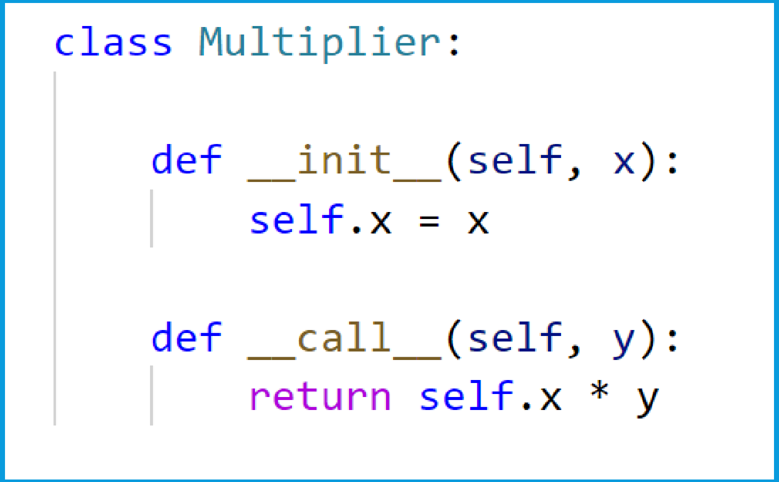
SOME THINGS HERE TO CONSIDER.

- _new_ is the first method called which creates you the object and passes to _init_ so your properties are initialized there.
- If the same attribute name occurs in both an instance and in a class, then attribute lookup prioritizes the instance.
- You can define private attributes with double underscore which is renamed to _classname__attrName.
- This is done to keep it hidden from out of the class and never use such variables in your code outside the class. (even if the class is yours)

CALLABLE CLASS METHOD

Aren’t all class callable ? What do you mean?

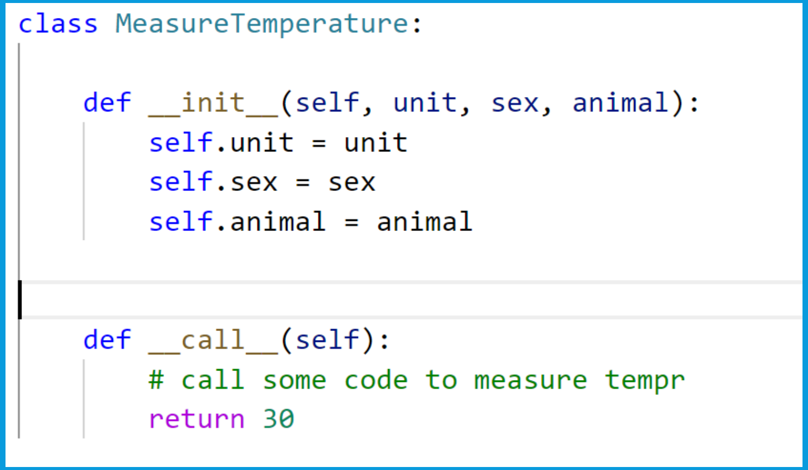
- Well yes but actually no.
- By callable class I mean the object can be made callable just by a simple dunder (magic) method.
- You can use wrapper functions to do same thing but classes are more flexible.



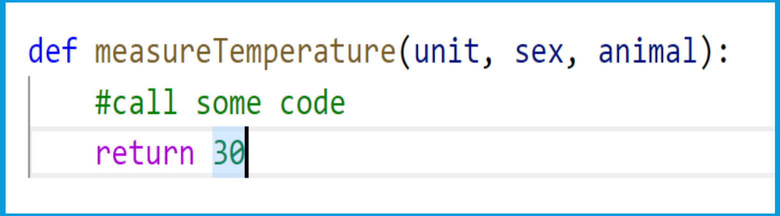
```python
class Multiplier:

    def __init__(self, x):
        self.x = x

    def __call__(self, y):
        return self.x * y
```

WHEN TO USE CALLABLE CLASS ?

- If your class has a certain function to cover.
- Your function is getting so messy you want to organize your function as a class to keep things simple.



```python
class MeasureTemperature:

    def __init__(self, unit, sex, animal):
        self.unit = unit
        self.sex = sex
        self.animal = animal


    def __call__(self):
        # call some code to measure tempr
        return 30
```



```python
def measureTemperature(unit, sex, animal):
    #call some code
    return 30
```

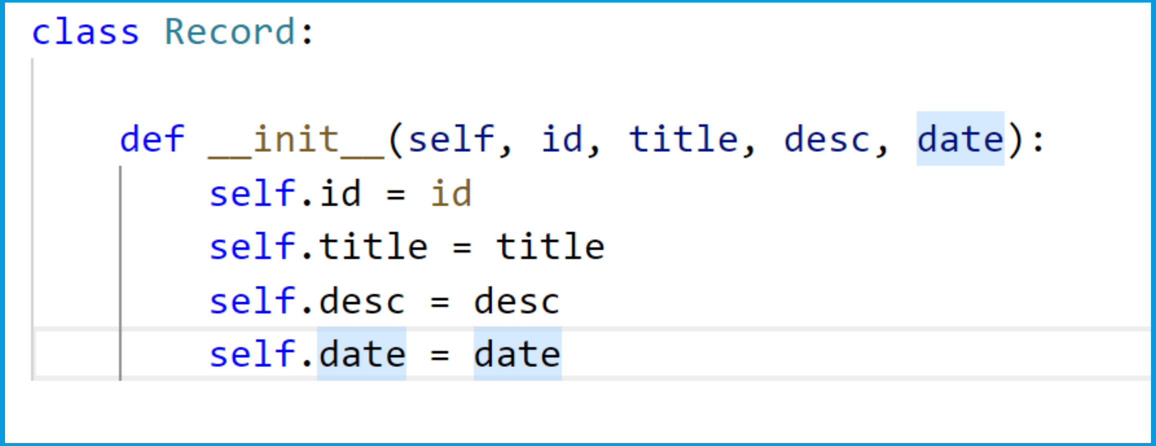
BUT THE FUNCTION LOOKS BETTER IN THIS CASE ?

- Think you're measuring ten thousands of animals as per the client's demand.
- Your program grows and you need to check if animal is dead, if animal is eating and much more. i.e. your program won't scale if you're using functions.

DATACLASSES.

- How many times have you used python classes to simulate behavior similar to struct in python?

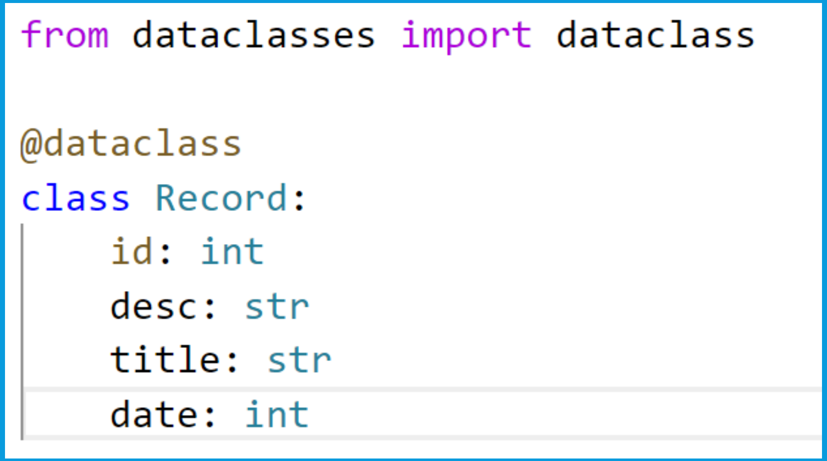
- I guess this is how you define it.



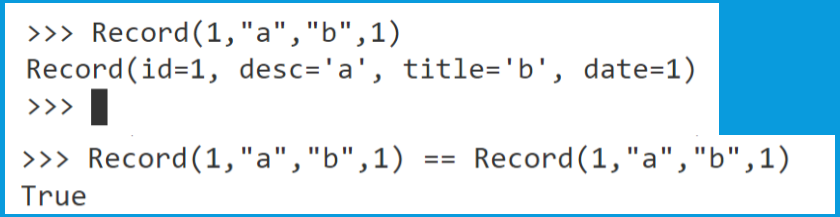
```python
class Record:

    def __init__(self, id, title, desc, date):
        self.id = id
        self.title = title
        self.desc = desc
        self.date = date
```

- It’s correct but that’s not what all you want.
- If you want to check if two records are equal python will always tell you no they’re not as two instances are never same unless you define _eq_ and customize how python compares yourself.
- When you add a new property to your class, you need to write code in both _init_ and initialize there.
- Your class won’t have good representation when you print you’ll get something like <object blahblah>

- Now when you make a variable, you will get a good representation and you can compare between two objects.



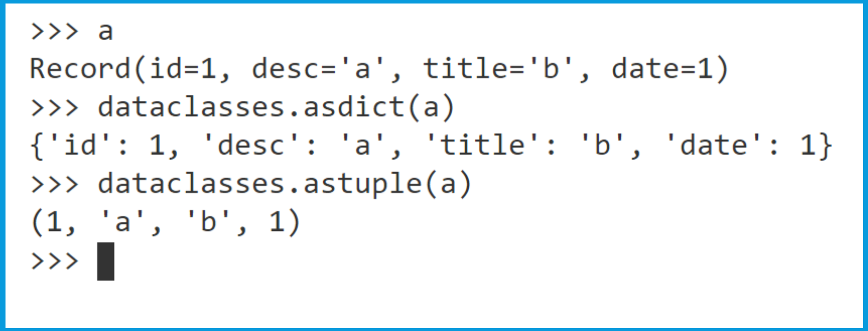
```python
from dataclasses import dataclass

@dataclass
class Record:
    id: int
    desc: str
    title: str
    date: int
```



```
>>> Record(1,"a","b",1)
Record(id=1, desc='a', title='b', date=1)
>>> 
>>> Record(1,"a","b",1) == Record(1,"a","b",1)
True
```

FURTHER MORE…

- This gets really useful when you are writing an ORM or structuring your data to be really flexible and strict at same time.
- Keep in mind, these are also just a plain class you can have methods of a.
- Make sure don't write your own constructor or use _post_init_.



```
>>> a
Record(id=1, desc='a', title='b', date=1)
>>> dataclasses.asdict(a)
{'id': 1, 'desc': 'a', 'title': 'b', 'date': 1}
>>> dataclasses.astuple(a)
(1, 'a', 'b', 1)
>>> 
```

And we finish here

@PROPERTY

@PROPERTY DECORATOR

- Pythonic way of getters and setters.
- All the work of hiding the variable and providing methods for access is done by property.
- Don’t pollute your class namespace with get_property and set_property.
- Your property is overridden and getter and setter will always be called no matter however and wherever you access the property.
- You can use the method way, but decorator is lot simpler and efficient.

COLLECTIONS

DICT (BUT MORE THAN DICT)

- You know dict right ?
- It has a key-value pair. That’s a basic understanding of dict. Turns out python offers you some convenient varieties of dict.
- I’ve used defaultdict almost everywhere but most of the people have no idea what it is. (I’ll explain)
- Isn’t it cool ? How many times you’ve struggled to have a dictionary with every values as list, well not anymore.

DEQUE

- You all have worked with lists and you know the headache when you need to add items in front of the list frequently.
- Above problem can be solved with some functions but what if you also need to rotate your list frequently?
- Straight from the docs, **though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation.**

NAMEDTUPLE

- Just a well documented tuple.
- Isn’t it cool if each values of your tuple have their own names and also with flexibility of a normal tuple.
- It’s same as dataclasses but you don’t need to worry about making a whole new class just if you want your tuple to be documented.

THAT’S ALL ! :D

I hope you’ll utilize these things in your code from today and happy python-ing.

ENDING…

@regmicmahesh Cross Platform Full Stack Developer

Thank you so much!