Digital Career Institute

Python Course - Functions

Digital Career Institute

Goal of the Submodule

The goal of this submodule is to learn how to take advantage of functions to organize and optimize the code. By the end of this submodule, the learners will be able to understand:

- How to create and use decorators
- Lambda Functions

Topics

- Nested Callables
- Decorators
- Lambda Functions

Digital Career Institute

Defining Inner (Enclosing) functions

Digital Career Institute

Defining Inner functions

Inside other functions

```
>>> def outer(bar):
...     def inner(foo):
...         print(“Inner”)
...     inner(bar[::-1])
...     print(“Outer”)
...
>>> outer(“hello”)
Inner
Outer
>>> inner(“hello”)
Traceback (most recent call last):
  File "using-functions-nested.py", line 8, in
<module>
    inner('hello')
NameError: name 'inner' is not defined
```

Functions can also be defined inside another function. They belong to its local scope.

In this case, **`inner()`** can only be called from within **`outer()`**.

The global scope has no access to **`inner()`**.

Defining Inner functions

Digital Career Institute

Nested functions

```
>>> def outer(bar):
...     def inner(foo):
...         print(“Inner bar”, bar)
...     inner(“Goodbye”)
...     print(“Outer foo”, foo)
...
>>> outer(“hello”)
Inner bar hello
Traceback (most recent call last):
  File "using-functions-nested.py", line 9, in
<module>
    outer('hello')
  File "using-functions-nested.py", line 7, in
outer
    print('Outer foo', foo)
NameError: name 'foo' is not defined
```

The `inner` function has access to variables in the global scope and also to those defined in the local scope of `outer()`.

The variable names defined in the local scope of `inner()` cannot be accessed by any instruction in neither the global scope or the scope of `outer()`.

Referring functions

Digital Career Institute

Referring functions

Digital Career Institute

*Python functions are **first-class citizens**.*

First-class citizens can be:

- assigned to variables
- stored in collections
- created and deleted dynamically
- passed as arguments

Referring functions

Digital Career Institute

As variables

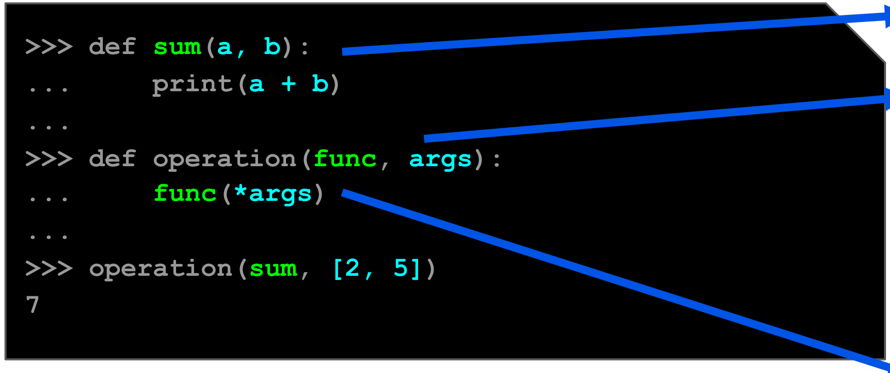
```
>>> def bar(bar):
...     print(bar)
...
>>> bar(“hello”)
hello
>>> my_alias = bar
>>> my_alias(“goodbye”)
goodbye
```

If we write the name of the function without the parenthesis we are not executing the function, but just **referring to it**.

`my_alias` dos not store the output of `bar`, it is just pointing to it and will behave likewise.

Referring functions

As input arguments



```
>>> def sum(a, b):
...     print(a + b)
...
>>> def operation(func, args):
...     func(*args)
...
>>> operation(sum, [2, 5])
7
```

`sum()` adds the value of two numbers.

`operation()` takes a function and a list of arguments.

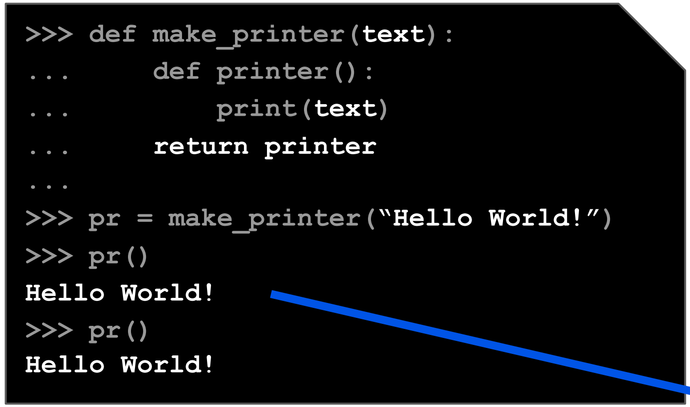
The "alias" of that function in the local scope of **`operation`** is **`func`**.

It unpacks the **`args`** list to pass it as individual arguments to the call to **`func()`**, which in this case is an alias of **`sum()`**.

Functions that take other functions as arguments are called ***higher order functions***.

Closures

As output arguments



```
>>> def make_printer(text):
...     def printer():
...         print(text)
...     return printer
...
>>> pr = make_printer("Hello World!")
>>> pr()
Hello World!
>>> pr()
Hello World!
```

A closure is a function that returns a function, who is using a variable from the first function.

- `make_printer()` defines and returns a function that is using a resource from the scope of `make_printer()`.
- It **returns a function** that we get in our global scope into a variable.
- We call the function and it prints the value of the variable named `text`.

The literal `"Hello World!"` is not stored anywhere other than `text` and `make_printer()` has finished the execution but the value of `text` did not disappear and we still can print it.

Decorators

Digital Career Institute

Decorators

Digital Career Institute

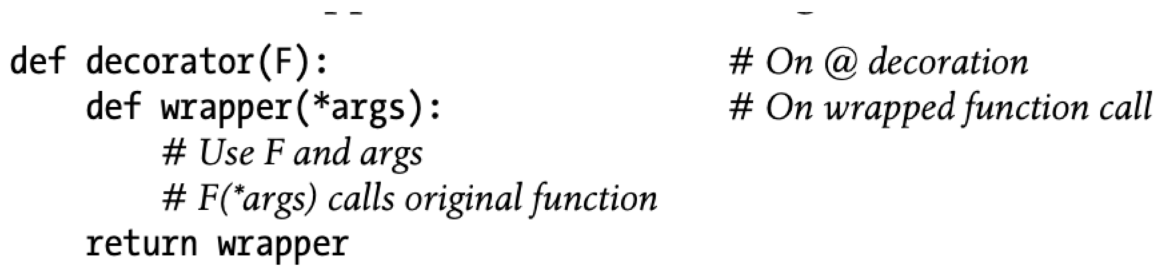
They give additional functionality to our existing functions without modifying them.

Decorators

They are a way to specify Management code and Augmentation code for a function or class.

It is a way to insert automatically run code at the end of a function or class definition.

Decorators, How to create one



```
def decorator(F):                       # On @ decoration
    def wrapper(*args):                 # On wrapped function call
        # Use F and args
        # F(*args) calls original function
    return wrapper
```

Decorators, How to use one

```
@decorator                  # Decorate function
def F(arg):
    ...

F(99)                       # Call function
```

What the Above means

```
F = decorator(F)            # Rebind function name to decorator result

F(99)                       # Essentially calls decorator(F)(99)
```

Decorators

*A closure that **uses a function** as an argument and executes it (or not) in the enclosed function. Callable that return another callable*

```
>>> def make_pretty(func):
...     def inner():
...         print(“I’m decorated”)
...         func()
...     return inner
...
>>> def ordinary():
...     print(“Hello World!”)
...
>>> ordinary()
Hello World!
>>> pretty = make_pretty(ordinary)
>>> pretty()
I’m decorated
Hello World!
```

`make_pretty()` is a decorator that prints some text before executing the function passed to it.

Decorators are a way to add a specific functionality to our functions. They are also, themselves, functions.

We can apply the decorator temporarily into a new variable name simply by calling it and passing the function we want to decorate.

Decorators

Shortcut

```
>>> def make_uppercase(func):
...     def inner():
...         return func().upper()
...     return inner
...
>>> @make_uppercase
>>> def greeting():
...     return "Hello World!"
...
>>> print( greeting() )
HELLO WORLD!
```

Our functions can be permanently decorated by preceding their definitions with **`@name_decorator`**.

This is equivalent to doing **`greeting = make_uppercase(greeting)`** right after the definition of **`greeting`**.

Decorators

Multiple decorators

```
>>> def make_count(func):
...     def inner():
...         return str(len(func()))
...     return inner
...
>>> def make_split(func):
...     def inner():
...         return func().split()
...     return inner
...
>>> @make_count
>>> @make_split
>>> def ordinary():
...     return "Hello World!"
...
>>> print( ordinary() )
2
```

The decorators will be applied starting from the closest to the definition, in this case `@make_split`.

We first split the sentence into a list of words and then count the resulting list. Our sentence has `2` words.

Inverting the decorators will count the characters in the string, convert them to text and split that text into a list, returning `['12']`.

Decorators

Using function arguments

```
>>> def make_capitalized_arguments(func):
...     def inner(*args):
...         capitalized = [w.capitalize()
...                        for w in args]
...         return func(*capitalized)
...     return inner
...
>>> @make_capitalized_arguments
>>> def greeting(first, last):
...     return f"Hello, {first} {last}!"
...
>>> print( greeting("jAMES", "bROWN") )
Hello, James Brown!
```

If the function to be decorated has parameters we must include them in our decorator's `inner` function and include them in the `func` call.

We can operate, or not, with those arguments.

Decorators

Using decorator arguments

```
>>> def make_capitalized_arguments(*deco_args):
...     def decorator(func):
...         def inner(**kwargs):
...             data = []
...             for k, w in kwargs.items():
...                 if k in deco_args:
...                     data.append(w.capitalize())
...                 else:
...                     data.append(w)
...             return func(*data)
...         return inner
...     return decorator
...
>>> @make_capitalized_arguments("first")
>>> def greeting(first, last):
...     return f"Hello, {first} {last}!"
...
>>> print( greeting(first="jAMES", last="bROWN") )
Hello, James bROWN!
```

We need to wrap another function around to hold the arguments of the decorator.

And return the inner `decorator`.

We can now specify which of the function keyword arguments we want to capitalize.

Lambda functions

Digital Career Institute

Defining functions

Lambda functions

Lambda functions are special **anonymous functions** defined as **expressions**.

```
>>> add1 = lambda x: x + 1
>>> print(add1(1))
2
>>> # This is equivalent to
>>> def add1(x):
...     return x + 1
...
>>> print(add1(1))
2
```

They are anonymous, but they are still *first-class citizens* and can be assigned to a variable.

They have their own syntax.

- They use **`lambda`** instead of **`def`**.
- They do not have a name.
- They do not need parentheses.
- They do not use the **`return`** keyword.
- They can’t use multiple lines.

Defining functions

Lambda functions

```
>>> add1 = lambda x: x + 1
```

Input parameters. There can be any number or 0.

Output parameter.

Defining functions

Lambda functions

```
>>> multiply = lambda x, y: x * y
>>> print(multiply(1, 0))
0
>>> print(multiply(3, 9))
27
>>> def printer(bar):
...     return lambda x: f"{bar}, {x}!"
...
>>> greet = printer("Hello")
>>> print(greet("John"))
Hello, John!
```

They can take any number of arguments.

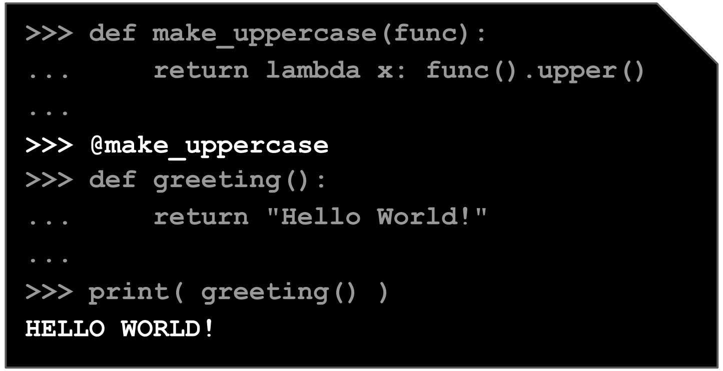
They can take any number of arguments.

They help keep the code a little more concise.

Defining a lambda function and assigning it to a variable name right away is not recommended by the PEP-8 style guide.

Decorators

Lambda functions



```
>>> def make_uppercase(func):
...     return lambda x: func().upper()
...
>>> @make_uppercase
>>> def greeting():
...     return "Hello World!"
...
>>> print( greeting() )
HELLO WORLD!
```

Often used in **closures** and **decorators** or as arguments to be passed to *higher-order functions*.

They help keep the code a little more concise and prevents the creation of a variable name that is not required.

We learned …

- That we can call functions from inside another function as well as from the same function.
- That recursive functions need to identify a base case to halt the recursion.
- That they operate in two steps: drill-down and backwards calculations.
- That functions can be nested and contained within other functions.
- That functions are first-class citizens and can be assigned to variables and passed as arguments.
- That closures are functions that return a function and this one has access to the first function scope.
- That decorators are closures that take a function as an argument.
- How to work with function and decorator arguments.
- How to create anonymous functions by using lambda functions.

Digital Career Institute

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute
DCI