CS1302 - Lecture 6 More Functions – (b2) Arguments and Decorator
(Dr Helena WONG)

==========

[Contents]

==========

[Helena's supplements - Exercises]

Starter Exercises

a) Guess the outcomes from the program below:

```
def calculate_sum(x=1, y=2):
    return x+y

print(calculate_sum())      # ________
print(calculate_sum(10))    # ________
print(calculate_sum(10,20)) # ________
```

b) Guess the outcomes from the program below:

```
def calculate_sum(x,y=5): # x is a required argument, but y is optional
    return x+y

print(calculate_sum(1,8))     # _____
print(calculate_sum(1))       # _____
print(calculate_sum(x=1,y=8)) # _____
print(calculate_sum(x=1))     # _____
print(calculate_sum(y=1,x=8)) # _____
```

==========

Arguments and Decorator

1. Optional Arguments

Keyword argument vs positional argument
Arguments can be required **or** optional (if given default value / "default argument")
Rules
range(start, stop, step) - Takes only positional arguments (no keyword arguments)

2. Variable number of arguments

Calling a function with an arbitrary number of arguments
The function to get both multiple non-keyword and keyword arguments
Writing fibonacci like range

3. Decorator

What is function decoration
Why decorate a function
Why use a variable number of arguments in wrapper
Why decorate the wrapper with @functools.wraps(f)?

4. Module

recurtools.py and decorating a recursive function (factorial)

==========

1. Optional arguments

Two types of arguments:

keyword arguments:

```
c1 = complex(real=3, imag=5)
c2 = complex(imag=5, real=3)
```

positional arguments:

```
c3 = complex(3, 5)
```

Arguments can be required **or** optional (if given default value)

Basic: Arguments are required if we do not provide the default

```
def calculate_sum(x,y):
    return x+y
print(calculate_sum(1,8))     # give positional arguments
print(calculate_sum(x=1,y=8)) # give keyword arguments
print(calculate_sum(y=1,x=8)) # give keyword arguments
```

Optional is made possible:

Arguments are optional if we provide the default

```
------------
def calculate_sum(x=1, y=2):
    return x+y

print(calculate_sum())      # 3
print(calculate_sum(10))    # 12
print(calculate_sum(10,20)) # 30
------------
def calculate_sum(x,y=5):     # x is a required argument, y is optional
    return x+y

print(calculate_sum(1,8))     # 9
print(calculate_sum(1))       # 6
print(calculate_sum(x=1,y=8)) # 9
print(calculate_sum(x=1))     # 6
print(calculate_sum(y=1,x=8)) # 9
------------
```

Rules for function definition:

```
def my_sum(x=3, y): # SyntaxError: non-default argument follows default argument
    return x+y
```

Rules for calling functions:

1. Keyword arguments must be after all positional arguments (wrong: calculate_sum(y=8, 1))
2. Duplicate assignments to an argument are not allowed (wrong: calculate_sum(1, x=1)

```
def my_sum(x,y):
    return x+y

print(my_sum(x=10, 1)) # SyntaxError: Positional argument follows keyword argument
print(my_sum(1, x=1))  # TypeError:calculate_sum() got multiple values for argument 'x'
print(my_sum(1, 2))    # ok
print(my_sum(1, y=2))  # ok
print(my_sum(x=1, y=2))# ok
```

range(start, stop, step) -- a special case

- Takes only positional arguments (no keyword arguments)

```
r1 = range(1,10,2)        # OK. range(start, stop[, step]) -> range object
r2 = range(1,10,step=2)  # TypeError: range() takes no keyword arguments
```

- Has default value for start **and** step**, but not stop**

```
r3 = range(1, 10, 2)      # tells start, stop, and step
r4 = range(1, 10)         # tells start, stop
r5 = range(10)            # tells stop!!
```

==========

2. Variable number of arguments

***args** (for positional arguments, received as a tuple)

****kwargs** (for keyword arguments, received as a dictionary)

tuple and dictionary

```
args = (0, 10, 2)  # a tuple
kwargs = {'start': 1, 'stop': 2, 'keyword': 6}  # a dictionary
```

Example 1: *args

```
def myFun(*args):
    print(args)
    print(type(args))
    for arg in args:
        print(arg)

myFun('Hello', 'Welcome', 'to', 'CS1302', 5, 10)
```

Example 1b: *args

```
def calculate_sum(*args):
    sum = 0
    for x in args:
        sum += x
    return sum

print(calculate_sum(1,2,3)) # 6
```

Example 2: **kwargs

```
def myFun(**kwargs):
    print(kwargs)
    print(type(kwargs))
    print(kwargs.keys())
    print(kwargs.values())
    print(kwargs.items())

    for key,value in kwargs.items():
        print(key,"=",value, end=" ")

myFun(mary="A", peter="B") # mary = A peter = B
```

The function to get both multiple positional (non-keyword) and keyword arguments

```
def myFun(*args, **kwargs):
    print(args)
    print(kwargs)

myFun(1,2,3,4,'add oil!', mary="A", peter="B")
```

More:

A function to return a string containing all arguments

def argument_string(*args, **kwargs) # see the lecture notebook:

```
def argument_string(*args, **kwargs):
    """Return the string representation of the list of arguments."""
    return "({})".format(
        ", ".join(
            [
                *["{!r}".format(v) for v in args],  # arguments
                *[
                    "{}={!r}".format(k, v) for k, v in kwargs.items()
                ],  # keyword arguments
            ]
        )
    )


argument_string(0, 10, 2, start=1, stop=2)
```

```
'(0, 10, 2, start=1, stop=2)'
```

Revising fibonacci: like range() - positional arguments depends on the count of arguments

```
def fibonacci_sequence(*args):

    Fn, Fn1, stop = 0, 1, None  # default values

    if len(args) == 1:
        stop = args[0]
    elif len(args) == 2:
        Fn, Fn1 = args[0], args[1]
    elif len(args) > 2:
        Fn, Fn1, stop = args[0], args[1], args[2]

    while stop is None or Fn < stop:
        yield Fn
        Fn, Fn1 = Fn1, Fn1 + Fn

r1=[*fibonacci_sequence(5,8,100)]
r2=[*fibonacci_sequence(10)]

g = fibonacci_sequence(5,8)
r3=[next (g) for i in range(5)]

# The above is a simplified version.
# If you read the one in the lecture notebook, it is longer because
# it tries to handle values passed by send: "value = yield Fn" etc..
#     Ref: Please refer to our teaching about Generator
# We don't repeat here because today we just focus on handling function arguments.
```

Naming: *args and **kwargs

args and kwargs are recommended names.

You may use other names but not recommended.

Keyword-Only Arguments

Keyword-Only Arguments (Bare `*`)

When a bare asterisk (without a parameter name immediately following) appears in the function definition's parameter list, it indicates that all parameters defined **after** it must be passed as keyword arguments. This makes the function call more readable by explicitly naming optional parameters. Stack Overflow +1

```
def user_info(first_name, last_name, *, age, level):
    print(f"Name: {first_name} {last_name}, Age: {age}, Level: {level}")

# Valid call
user_info("Jane", "Doe", age=30, level="Intermediate")
```

==========

3. Decorator

"a decorator is a function that takes another function and
extends the behavior of the latter function without explicitly modifying it."

A very simple introduction (the wrapped function takes no argument)

```
def f1(func):
    def wrapper():
        print("-----------")
        func()
        print("===========")
    return wrapper

def f():
    print("Hello")

# w = f1(f)
# w()

f = f1(f)
f()

# print(f.__name__) # "f" or "wrapper"?
```

```
def f1(func):
    def wrapper():
        print("-----------")
        func()
        print("===========")
    return wrapper

def f():
    print("Hello")

# w = f1(f)
# w()

f = f1(f)
f()
```

```
-----------
Hello
===========
```

```
def f1(func):
    def wrapper():
        print("-----------")
        func()
        print("===========")
    return wrapper

@f1
def f():
    print("Hi")

f()
```

```
-----------
Hi
===========
```

Learn:

* Use a variable number of arguments in wrapper

* Add the return statement in wrapper

* Decorate the wrapper with @functools.wraps(func); requires *import functools*

  Add useful attributes. E.g., __wrapped__ stores the original function so we can undo the decoration.

  Ensures some attributes (such as __name__) of the wrapper function is the same as those of f.
  (Ref https://www.freecodecamp.org/news/whats-in-a-python-s-name-506262fe61e8/)

* __wrapped__: restore the original function (restore to the one before decoration)

==========

Example 1 - (i) Multiple arguments, (ii) Return value:

==========

```
import random

def my_decorator(func):
    def wrapper(*args,**kwargs):
        print("-----------")
        value = func(*args,**kwargs)
        print("===========")
        return value
    return wrapper

@my_decorator
def g(a,b):
    print("Lucky number:", random.randint(a,b))

@my_decorator
def g1(a,b):
    return random.randint(a,b)

g(100,200)
x = g1(100,200)
```

==========

Example 2 **functools.wraps(..)** : (i) __name__ atribute, (ii) __wrapped__ (original function)

==========

```
import random, functools

def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args,**kwargs):
        print("-----------")
        value = func(*args,**kwargs)
        print("===========")
        return value
    return wrapper

@my_decorator
def f():
    print("Hello")

@my_decorator
def g(a,b):
    print("Lucky number:", random.randint(a,b))

@my_decorator
def g1(a,b):
    return random.randint(a,b)

f()
g(100,200)
x = g1(100,200)
n1 = f.__name__
n2 = g.__name__
n3 = g1.__name__

f = f.__wrapped__
f()
```

==========

The **count_prime_nums** example

==========

```
import functools

def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args,**kwargs):
        print("started")
        value = func(*args,**kwargs)
        print("ended")
        return value
    return wrapper

def is_prime(num):
    if num<2:
        return False
    elif num==2:
        return True
    else:
        for i in range(2,num):
            if num%i ==0:
                return False
        return True

@my_decorator
def count_prime_nums(x,y):
    count=0
    for i in range(x,y):
        if is_prime(i):
            count+=1
    print("there are",count,"prime numbers between", x, "and", y)

count_prime_nums(2,10)
count_prime_nums.__wrapped__(2,20)

count_prime_nums = count_prime_nums.__wrapped__
count_prime_nums(2,30)
```

==========

3. Module

We can create a module: put the code in a Python source file <module name>.py in
the current directory, or
a Python site-packages directory in system path. (See the Lecture notebook)

```
===============
Example 1: print_function_call
===============
```

```
        import recurtools as rc

        @rc.print_function_call
        def factorial(n):
            if n>1:
                return n*factorial(n-1)
            else:
                return 1

        r1 = factorial(5)
        print("result: ", r1, end="\n\n")

        r2 = factorial(6)
        print("result: ", r2, end="\n\n")

        r3 = factorial(7)
        print("result: ", r3, end="\n\n")
```

```
===============
Example 2: caching + clear_cache()
===============
```

```
        import recurtools as rc
        import functools

        @rc.caching
        @rc.print_function_call
        def factorial(n):
            if n>1:
                return n*factorial(n-1)
            else:
                return 1


        r1 = factorial(5)
        print("result: ", r1, end="\n\n")

        r2 = factorial(6)
        print("result: ", r2, end="\n\n")

        functools.update_wrapper(factorial, factorial.__wrapped__, assigned=('cache_clear',))
        factorial.cache_clear()
        r3 = factorial(7)
        print("result: ", r3, end="\n\n")
```

==========

See additional contents from **Generator and Decorator.ipynb**