More on functions

In this lecture, we also various important features of designing functions in Python

Created by John C.S. Lui on May 31, 2018.

Important note: If you want to use and modify this notebook file, please acknowledge the author.

In [ ]:
```
# Let's define a display function

def display_func(name):
    print ("To " + name + ", ", end='')
    print ("welcome to CSCI2040.")

display_func('John')
display_func('Jack')

# calling the function below will create a problem
#display_func()
```

Setting default values for argument to a function

We can avoid the above problem if we have some default values. Let's see.

In [ ]:
```
# Let's define a display function with default value

def display_func(name='nobody'):
    print ("To " + name + ", ", end='')
    print ("welcome to CSCI2040.")

display_func('John')
display_func('Jack')
display_func()
```

Positional arguments in function

In [ ]:
```
# This function takes in a person's first and last name, his/her age and prints this information out

def describe_person(first_name, last_name, age):

    print("First name: %s" % first_name.title())
    print("Last name: %s" % last_name.title())
    print("Age: %d\n" % age)

describe_person('brian', 'kernighan', 71)
describe_person('ken', 'thompson', 70)
describe_person('john c.s.', 'lui', 28)
```

Comment for the above program

For the above program, *first_name*, *last_name*, and *age*. These are called **positional arguments**,
Python knows which value to assign to each by the order in which you give the function values.

What if we want to pass these arguments in whatever order? Let's illustrate the concept of **keyword argument**.

In [ ]:
```
def describe_person(first_name, last_name, age):
    print("First name: %s" % first_name.title())
    print("Last name: %s" % last_name.title())
    print("Age: %d\n" % age)

describe_person(age=71, first_name='brian', last_name='kernighan')    # using keyword argument format
describe_person(age=70, first_name='ken', last_name='thompson')
describe_person(age=68, first_name='John C.S.', last_name='Lui')
```

Mixing positional and keyword arguments and default values

Let's see.

In [ ]:
```
def describe_person(first_name, last_name, age=None, favorite_language=None, died=None):
    # Required information:
    print("First name: %s" % first_name.title())
    print("Last name: %s" % last_name.title())
    # Also initialize some arguemnts with default values

    # Optional information:
    if age:
        print("Age: %d" % age)
    if favorite_language:
        print("Favorite language: %s" % favorite_language)
    if died:
        print("Died: %d" % died)

    # Blank line at end.
    print("\n")

describe_person('brian', 'kernighan', favorite_language='C')
describe_person('ken', 'thompson', age=70)
describe_person('adele', 'goldberg', age=68, favorite_language='Smalltalk')
describe_person('dennis', 'ritchie', favorite_language='C', died=2011)
describe_person('guido', 'van rossum', favorite_language='Python')
```

Function which takes on an arbitrary number of arguments

We have now seen that using *keyword arguments* can allow for much more flexible calling statements.

- This benefits you in your own programs, because you can write one function that can handle many different situations you might encounter.
- This benefits you if other programmers use your programs, because your functions can apply to a wide range of situations.
- This benefits you when you use other programmers' functions, because their functions can apply to many situations you will care about.

There is another issue that we can address, though. Let's consider a function that takes two number in, and prints out the sum of the two numbers:

In [ ]:
```
# This function adds two numbers together, and prints the sum
def adder(num_1, num_2):
    sum = num_1 + num_2
    print("The sum of your numbers is %d." % sum)

# Let's add some numbers.
adder(1, 2)
adder(-1, 2)
adder(1, -2)
```

This function appears to work well. But what if we pass it three numbers, which is a perfectly reasonable thing to do mathematically?

In [ ]:
```
def adder(num_1, num_2):
    sum = num_1 + num_2
    print("The sum of your numbers is %d." % sum)

# Let's add some numbers.
adder(1, 2)
adder(1, 2, 3)
```

Let's try to fix this problem. If we place an argument at the end of the list of arguments, with an asterisk in front of it, that argument will collect any remaining values from the calling statement into a tuple. Here is an example demonstrating how this works:

In [ ]:
```
# define a new function and let's pay attention at the argument values.
def example_function(arg_1, arg_2, *arg_3):   # see the *arg_3 ?
    print('\narg_1:', arg_1)
    print('arg_2:', arg_2)
    print('arg_3:', arg_3)

example_function(1, 2)
example_function(1, 2, 3)
example_function(1, 2, 3, 4)
example_function(1, 2, 3, 4, 5)
```

We can use a *loop* to process these other arguments.

In [ ]:
```
def example_function(arg_1, arg_2, *arg_3):
    print('\narg_1:', arg_1)
    print('arg_2:', arg_2)
    for value in arg_3:
        print('arg_3 value:', value)

example_function(1, 2)
example_function(1, 2, 3)
example_function(1, 2, 3, 4)
example_function(1, 2, 3, 4, 5)
```

Let's rewrite our *adder()* function

In [ ]:
```
# This function adds the given numbers together, and prints the sum.

def adder(num_1, num_2, *nums):

    sum = num_1 + num_2    # Start by adding the first two numbers, which will always be present.

    for num in nums:       # Then add any other numbers that were sent.
        sum = sum + num

    print("The sum of your numbers is %d." % sum)    # Print the results.

# Let's add some numbers.
adder(1, 2, 3)
adder(1,2,3,4,5)
```

Accepting an arbitrary number of keyword arguments

Python also provides a syntax for accepting an arbitrary number of keyword arguments. The syntax looks like this:

In [ ]:
```
# Let's look at the argument values.
def example_function(arg_1, arg_2, **kwargs):
    print('\narg_1:', arg_1)
    print('arg_2:', arg_2)
    print('arg_3:', kwargs)

example_function('a', 'b')
example_function('a', 'b', value_3='c')
example_function('a', 'b', value_3='c', value_4='d')
example_function('a', 'b', value_3='c', value_4='d', value_5='e')
```

The third argument has two asterisks in front of it, which tells Python to collect all remaining key-value arguments in the calling statement. This argument is commonly named **kwargs**. We see in the output that these key-values are stored in a dictionary. We can **loop** through this dictionary to work with all of the values that are passed into the function:

In [ ]:
```
def example_function(arg_1, arg_2, **kwargs):
    print('\narg_1:', arg_1)
    print('arg_2:', arg_2)
    for key, value in kwargs.items():
        print('arg_3 value:', value)

example_function('a', 'b')
example_function('a', 'b', value_3='c')
example_function('a', 'b', value_3='c', value_4='d')
example_function('a', 'b', value_3='c', value_4='d', value_5='e')
```

Final example

In [ ]:
```
# This function takes in a person's first and last name, and an arbitrary number of keyword arguments.
def describe_person(first_name, last_name, **kwargs):

    # Required information:
    print("First name: %s" % first_name.title())
    print("Last name: %s" % last_name.title())

    # Optional information:
    for key in kwargs:
        print("%s: %s" % (key.title(), kwargs[key]))

    # Blank line at end.
    print("\n")

describe_person('brian', 'kernighan', favorite_language='C', famous_book='The C Programming Language')
describe_person('ken', 'thompson', age=70, alma_mater='UC Berkeley')
describe_person('adele', 'goldberg', age=68, favorite_language='Smalltalk')
describe_person('dennis', 'ritchie', favorite_language='C', died=2011, famous_book='The C Programming Langu
describe_person('guido', 'van rossum', favorite_language='Python', company='Dropbox')
```

In [ ]: