Assigning Parameters Default Values

When you use functions such as `range` and `print`, the number of arguments that you pass to them can vary. If you do not pass an argument for every parameter, then the default parameter values will be used. You will now learn how to define your own functions that use default parameter values.

The `print` Function

Let us first explore the `print` function:

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout)

    Prints the values to a steam, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (Stream); defaults to the current sys.stdout.
    sep: string inserted between values, default a space.
    end: string appended after the last value, default a newline.

>>>
```

Here we see that `print` has several parameters:

- `value, ...,`: the values to be printed.
- `sep=' '`: an optional argument that by default will be a space. When multiple values are printed, this string will be printed between pairs of values.
- `end='\n'`: an optional argument that by default will be a newline character. This string is printed after the last value.
- `file`: an optional argument that by default is `sys.stdout`, which specifies where to print.

Let us now use the `print` function to learn more about its behaviour:

```
>>> print('345')
345
>>>
```

We see that the newline character, `'\n'`, is automatically printed after value `'345'`.

```
>>> print(1, 2, 3)
1 2 3
>>>
```

Here, since we printed multiple values, the default separator (a space) is printed in between each value being printed. Once more, the newline character (`'\n'`) is printed at the end. We will use `print` again, but this time we'll pass arguments to use instead of the default parameter values:

```
>>> print(1, 2, 3, sep='..', end='!')
1..2..3!
>>>
```

We see that the arguments that we passed to the function call were used in place of the default parameter values.

Using Default Parameter Values in Our Own Function

Let us now define a function with default parameters values. We will define a function called `every_nth` that takes two arguments. The first argument will be a `list` and the second argument will be an `int`. The job of this new function will be to create a new list `L` that contains every `n`th item from the list `L`. Here is the definition of function `every_nth`:

```
def every_nth(L, n):
    """ (list, int) -> list

    Precondition: 0 <= n < len(L)

    Return a list containing every nth item of L,
    starting at index 0.

    >>> every_nth([1, 2, 3, 4, 5, 6], 2)
    [1, 3, 5]
    >>> every_nth([1, 2, 3, 4, 5, 6], 3)
    [1, 4]
    """

    result = []

    for i in range(0, len(L), n):
        result.append(L[i])

    return result


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

We have not introduced default parameter values yet. At the moment, both of the example function calls pass in two arguments. We will now modify the `docstring` and the function header to allow for a default parameter value to be used for the second parameter.

```
def every_nth(L, n=1):
    """ (list, int) -> list

    Precondition: 0 <= n < len(L)

    Return a list containing every nth item of L,
    starting at index 0.

    >>> every_nth([1, 2, 3, 4, 5, 6], 2)
    [1, 3, 5]
    >>> every_nth([1, 2, 3, 4, 5, 6], 3)
    [1, 4]
    >>> every_nth([1, 2, 3, 4, 5, 6])
    [1, 2, 3, 4, 5, 6]
    """

    result = []

    for i in range(0, len(L), n):
        result.append(L[i])

    return result


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

We see that if we run the above code, the `doctest` passes. When passing the second argument, we can also specify which parameter it is being associated with (e.g., `every_nth([1, 2, 3, 4, 5, 6], n=3)`). With functions that have multiple default parameter values, using this notation is necessary in order to associate the arguments with the correct parameters.

More on Default Parameters

Let us now look at more examples of functions using default parameter values.

```
def add_greeting(L=[]):
    """ (list) -> NoneType

    Append 'hello' to L and print L.

    >>> L = ['hi', 'bonjour']
    >>> f(L)
    >>> L
    ['hi', 'bonjour', 'hello']
    """

    L.append('hello')
    print(L)
```

In this case, the default parameter value is mutable. Here, we have just one parameter, and it has the default value of an empty list `L=[]`. This function simply appends the string `'hello'` to the list, and then prints the list. Let us see what happens when we call this function:

```
>>> add_greeting()
['hello']
```

We see that the string `'hello'` is added to the list and then printed. Let us see what happens when the function is called multiple times:

```
>>> add_greeting()
['hello']
>>> add_greeting()
['hello', 'hello']
>>> add_greeting()
['hello', 'hello', 'hello']
```

Here, we see that the same list object is being used for each function call. In other words, the memory address that `L` refers to is the same for each funtion call, so the string `'hello'` is appended to that same list each time the function is called.

The lesson to be learned here is that default parameter values are assigned at the time of function definition. When the function is called, any changes that the function call makes to the mutable object persist from one function call to the next.

Jennifer Campbell • Paul Gries
University of Toronto
