Managing Infinite Parameters

Did you ever wonder how the `print` function is written? Every function we have written so far has a *finite* number of parameters:

```
def add_me(a, b, c, d):
  return a+b+c+d
```

Calling `add_me` will **ONLY** work with four parameters. Yet with the print function you can call it with any number of parameters:

```
print('a')
print('a', 'b', 'c')
```

There is a special Python operator (the $*$) that you can use before a function parameter that means 'any number of'. You can create a function like `print` to accept any number of parameters by prefacing the parameter name with the $*$:

```
def add_me(*args):
  v = 0
  for e in args:
    v += e  # same as v = v + e
  return v

print(add_me(1,2))
print(add_me(1,2,3,4))
$_
```

The name **`args`** is by convention, but any legal variable name will work.

Closely related to the $*$ syntax to define a function that takes any number of parameters is the $**$ syntax that can be used to define a function that takes any number of key/value pair arguments (just like a dictionary):

```
def name_value_pairs(**kargs):

  # access kargs like a dictionary
  for k,v in kargs.items():
    print(k,v)

  # just the keys please
  for k in kargs:
    print(kargs[k])

  # ask for a specific value
  print(kargs['classname'])
```

Exercise: Create the above function and then test it:

```
name_value_pairs(classname = 'Info 490',
                 credits = 3,
                 on_line = True)
```

Unpacking the argument

With all that in mind you can use the same * syntax to unpack a list to send the individual items of the list to a function that accepts an infinite number of arguments. Type the following code in to demonstrate it.

```
def simple_unpack_demo():
    numbers = [x for x in range(1,10)]
    print(numbers)
    print(add_me(*numbers))
```

In the above example *numbers is taking the list and unpacking it into its elements (1, 2, 3, 4, .. 9) and sending those individual elements to `add_me()`, with each element as its own parameter.

Similarly, you can use the ** syntax to unpack a dictionary to send it to a function that accepts an infinite number of key/value pair arguments:

```
def who_am_i(**kargs):
  for k,v in kargs.items():
    print(k,v)

me = { 'classname': 'Info 490',
       'credits':    3,
       'on_line':    True}

print("1",  me)
print("2", *me) # pass in keys only

who_am_i(**me)
```

So the same syntax is used to define a function that can have an infinite number of parameters, the * and ** are used to 'unpack' lists and dictionaries to pass to those functions. The * operator is sometimes called *positional unpacking* (or *positional expansion*); the ** operator is sometimes called *keyword expansion*.

Unpacking Lists to Print

Both `print` and string's `format` method are `*args` functions that we just learned about. That means we can use * operator to unpack lists to send to those functions:

```
def demo_unpack_list():
    my_items = [1,2,3,"apple"]
    print(my_items)
    print(*my_items)
```

Make sure you understand the difference in the above two calls to print.

Formatting strings with named indices

As we saw in the lesson on string formatting (info490), we can unpack a set of values. In the next example we use the index specifier to grab the fourth item in a tuple:

```
def demo_format1():
    values = (10,11,12,13)
    print('{3:04d}'.format(*values))
```

In addition to numeric indices, you can use names to reference variables passed into the format:

```
def demo_format2():
    info = "Info 490"
    str  = "Hello {name}!".format(name=info)
    print(str)
```

Note that since the parameter inside the format string references name (i.e. {name}), you must pass a parameter to `format` with that 'name'.

One **important** note is that if you use the $*$ for the first argument with format, you **MUST** use named parameters for the others.

```
def demo_format3():
    values = (10,11,12,13)
    answer = 12
    print('{2:04d} == {result:d}'.format(*values, result=answer))
```

Similarly you can use the $**$ syntax with the format method with dictionaries:

```
def demo_format4():
    me = {'name'    : 'Info 490',
          'credits': 3,
          'on-line': True}
    str = "Hello {name} you are worth {credits} credits!".format(**me)
    print(str)
```

Be sure to write and run both `demo_format3` and `demo_format4`.

Review Questions

1. What are positional arguments?
2. How ✱ and ✱✱ operate on lists and dictionaries?
3. What must you do if you pass in additional argument where the first argument is ✱?

Lesson Assignment:

Step 1.

Write a function named `multiply_me` which will multiply all of its parameters together.
So `multiply_me(1,2,3,4)` would return 24.
The function `multiply_me` can handle any number of parameters passed to it.

Step 2.

Create a function named `equation(numbers)`:

- `numbers` is a sequence of numbers
- builds the mathematical equation as a string that represents the result of `multiply_me`.
- it returns a string.

```
numbers = [x for x in
range(1,8)]
print(equation(numbers))

# The output would be:
1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040
```

The function `equation` would produce the string "`1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040`"

Similarly the following code:

```
numbers = [2 for x in range(5)]
print("2^5:", equation(numbers))
```

Would produce the following output: `2^5: 2 * 2 * 2 * 2 * 2 = 32`

Hints:

1. Get `multiply_me` working (e.g. `multiply_me(*numbers)`)
2. Remember `equation` builds a string using the `format` method
3. The same numbers passed into `multiply_me` are also passed into `format` (along with the result of `multiply_me`)
4. If you're writing a lot of code, try to re-think your solution

The End!

Material from Michael Haberman has been used!