1 More about Python function parameters and arguments

1.1 What you already know

We've used the term "parameters" to mean both the names used in function definitions and the values passed in a function call. Let's define more precise terminology:

- *Parameters* are the placeholders used in function *definitions*.
- *Arguments* are the values you give when you *call* a function.

Q. In the program below, what are the parameters and what are the arguments?

```
def f(a, b):
    print(a, b)

y = 4
f(12, y)
f(y, y)
```

A.

Our topic here is other ways of specifying both parameters and arguments. Unlike the basic ideas of functions, parameters and arguments, the programming techniques described here are not generally available in other programming languages.

1.2 Default parameter values

What if, a great deal of the time, one of the parameters of a function has some particular value? For example, suppose we have a function to add the numbers in a list:

```
def sum(L, base):
    total = base
    for item in L:
        total += item
    return total
```

Sometimes, you might want to have a non-zero value for the `base` parameter, for example when carrying over some previous total. However, mostly, you'd want to get the effect of calling "`sum(L, 0)`", and you'd be annoyed at having to type "`, 0`" all the time.

You can tell Python to assume `base` is 0 unless told otherwise:

```
def sum(L, base=0):
    total = base
    for item in L:
        total += item
    return total
```

```
print(sum([1, 2, 4]))  # prints ... what?


print(sum([1, 2, 4], 5))  # prints ... what?
```

1.3 Named arguments

You have to give names to parameters; here we're talking about having the *caller* use the *parameter* names to identify *arguments*.

You can give the arguments in any order, provided you name them:

```
print(sum(base=2, L=[6, 7])) # prints what?
```

You can mix *named* and *positional* arguments. (Positional arguments are the ones you're used to, given in the same order as the corresponding parameters.)

```
def func(a, b, c):
    print(a, b, c)

func(1, 2, 3)
func(b=5, a=1, c=3)
```

After the first named argument, all the rest must be named: `func(b=5, a=1, 3)` is illegal.

Q. Why?
A.

Here's another example:

```
def many_defaults(a, b=6, c='Toronto'):
    print(a, b, c)

many_defaults(9 % 5)

many_defaults(13 % 5, c='Ottawa') # b takes its default

many_defaults(c='Winnipeg', a=99)
```

Q. What's the output? (Write it with the function calls.)

1.4 Be careful with mutable default parameter values

```
def app(item, L=[]):
    L.append(item)
    return L

mylis = app('hi')
print(mylis)

otherlis = app('mom')
print(otherlis)
```

Q. What's the output?
A.

Q. Why?
A.

If you want the effect of always using the same mutable value as the default, use `None` as the default value instead:

```
def app(item, L=None):
    if L is None:
        L = []
    L.append(item)
    return L
```

1.5 Starred arguments

You can give a list in place of separate argument values:

```
def func(a, b, c):
    print(a, b, c)

L = ['Monday', 'Tuesday', 'Thursday']

func(L) # 1

func(*L) # 2

func(*[1, 2, 3, 4]) # 3
```

Q. What is printed by each function call? Why?

A.

You can have ordinary arguments before (but not after) a starred argument.

2 `break` and `continue`

Consider this program:

```
finished = False

while not finished:
    answer = input('Had enough? ')
    if answer == 'yes':
        finished = True
    elif answer != 'just a sec':
        print('One more pushup!')

print('All done!')
```

It's pretty clear, but the purpose of the input-and-response is to determine whether to keep on exercising. That might be clearer with the `break` and `continue` statements. Let's use them to rewrite the program.

Programmers disagree vociferously about these two statement types. They can be misused, but they do exist and sometimes can be helpful.

3 More on Functions

Functions are not just named chunks of code. They are also things that can be handled in most of the same ways as "ordinary" numerical, logical or string values.

- A variable can have a function as value.
- A function can be passed as a parameter.
- A function can be referred to by more than one variable as its value.

Let's write a function that takes a function as parameter.

```
def result_list(n, f, label):
    '''(int, function, str) -> NoneType

    Print label and then the result of applying f to the integers
    from 1 to n.  '''

    print(label, end = ' ')
    for i in range(n):
        print(f(i + 1), end = ' ')
    print()
```

3.1 map

`map` is a built-in that takes a function and a bunch of list-like things as parameters, like this:

```
>>> list(map(math.sqrt, [1, 2, 3, 4]))
[1.0, 1.4142135623730951, 1.7320508075688772, 2.0]
```

(Like `range`, `map` doesn't exactly return a list, but rather a "map object" that can be converted to a list.)

You can supply more than one set of values if the function you are mapping expects more than one argument:

```
>>> def add(i, j):
        return i + j

>>> list(map(add, [1, 2, 3, 4], range(10, 50, 10)))
```

Q. What's the output?

A.

3.2 Anonymous functions

If you just want to use a function once, and if its purpose is just to evaluate a one-line expression, you can use a ***lambda expression***:

```
(lambda x: x * x) (2)
```

It would be unusual to use a lambda expression like that, since it's simpler just to write "`2 * 2`". Here is a more plausible example:

```
>>> list(map(lambda x: x*x*x*x, range(1, 6)))
[1, 16, 81, 256, 625]
```

4 Using exceptions to handle errors

Exceptions are very simple, but because they change the usual "flow of control" — the sequence of statement execution — you need to think a little to figure them out.

An ***exception*** is a thing that Python creates to describe a problem it has detected, as in this excerpt from a Python shell session:

```
>>> 1/0
    Traceback (most recent call last):
      ...
    ZeroDivisionError: division by zero
```

The "`ZeroDivisionError`" is an exception.[2] Both its name and the message after its name help to tell the programmer what problem occurred.

What is especially interesting about the exception is what Python does with it. We say that the exception is ***raised***. When an exception has been raised, the entire progress of the program is derailed until the exception is ***handled***. If it is not handled, the running program is stopped and an error message is printed — just what happened to the Python command in the example.

4.1 Try-except blocks

How do you "handle an exception"? There is only one way: with an "`except` ***block***" (or "`except` clause"), which must follow a "`try` ***block***". The exception is *raised* in the `try` block and *handled* in the `except` block. Here's an example:

```
try:
    answer = input('An integer, please: ')
    i = int(answer)
    print('Thank you. Here are', i, 'candies.')
except:
    print("That wasn't an integer!")

print('This should be interesting ...')
print(i)
```

In the example, if the user gives an answer that cannot be converted to an `int`, then an exception is raised in the "`i = int(answer)`" statement — specifically, when the "`int`" function is called. ***The rest of the try block is ignored***, and the next statement to be executed is in the except block.

4.2 Handling more precisely

There can be many kinds of errors, and perhaps you don't want to handle them all in the same way. Here's an example where we simply give different error messages for different errors.

```
try:
    answer = input('An integer, please: ')
```

[2]Some exception types have the word "Exception" in their names, while others have "Error". The difference is not significant.

```
    i = int(answer)
    print('Thank you. Here are', i, 'candies.')
    print('It was good working with you,', name)
    i = i * 2
except ValueError:
    print("That wasn't an integer!")
except NameError:
    print('Unknown variable name')

print('This might be interesting ...')
print(i)
```

4.3 Raising exceptions yourself

If you detect trouble and want to escape immediately from whatever you're doing, you can raise an exception yourself, like this:

```
try:
    answer = input('? ')
    if answer == 'boo':
        raise Exception("I'm not at all happy.")
    print('Thank you for that', answer)
except Exception as e:
    print(e)
```

Here we used the standard, all-inclusive `Exception` type. It's possible to define your own types, but for that you need to know a little — not a lot — about object-oriented programming. Ask me if you are interested!