Declaring Functions

WE'LL COVER THE FOLLOWING

Python has functions like most other languages, but it does not have separate header files like `c++` or `interface/ implementation` sections like Pascal. When you need a function, just declare it, like this:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

The keyword `def` starts the function declaration, followed by the function name, followed by the arguments in parentheses. Multiple arguments are separated with commas.

Also note that the function doesn't define a return datatype. Python functions do not specify the datatype of their return value; they don't even specify whether or not they return a value. (In fact, every Python function returns a value; if the function ever executes a `return` statement, it will return that value, otherwise it will return `None`, the Python null value.)

> *In some languages, functions (that return a value) start with **function**, and subroutines (that do not return a value) start with **sub**. There are no subroutines in Python. Everything is a function, all functions return a*

> *value (even if it's **None**), and all functions start with **def**.*

When you need a function, just declare it.

The `approximate_size()` function takes the two arguments — `size` and `a_kilobyte_is_1024_bytes` — but neither argument specifies a datatype. In Python, variables are never explicitly typed. Python figures out what type a variable is and keeps track of it internally.

> *In Java and other statically-typed languages, you must specify the datatype of the function return value and each function argument. In Python, you never explicitly specify the datatype of anything. Based on what value you assign, Python keeps track of the datatype internally.*

Optional and Named Arguments

Python allows function arguments to have default values; if the function is called without the argument, the argument gets its default value. Furthermore, arguments can be specified in any order by using named arguments.

Let's take another look at that `approximate_size()` function declaration:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

The second argument, `a_kilobyte_is_1024_bytes`, specifies a default value of `True`. This means the argument is *optional*; you can call the function without it, and Python will act as if you had called it with `True` as a second parameter.

Now look at the bottom of the script:

```
if __name__ == '__main__':
    print(approximate_size(1000000000000, False))  # ①
    print(approximate_size(1000000000000))         # ②
```

① This calls the `approximate_size()` function with two arguments. Within the `approximate_size()` function, `a_kilobyte_is_1024_bytes` will be `False`, since you explicitly passed `False` as the second argument.

you explicitly passed `False` as the second argument.

② This calls the `approximate_size()` function with only one argument. But that's OK, because the second argument is optional! Since the caller doesn't specify, the second argument defaults to `True`, as defined by the function declaration.

You can also pass values into a function by name.

Let's run following 5 programs. Look at their outputs and then we can discuss each one of these.

Program 1

```
from humansize import approximate_size
print (approximate_size(4000, a_kilobyte_is_1024_bytes=False))       #①
#'4.0 KB'
```

① This calls the `approximate_size()` function with `4000` for the first argument (`size`) and `False` for the argument named `a_kilobyte_is_1024_bytes`. (That happens to be the second argument, but doesn't matter, as you'll see in a minute.)

Program 2

```
from humansize import approximate_size
print(approximate_size(size=4000, a_kilobyte_is_1024_bytes=False))   #②
#'4.0 KB'
```

② This calls the `approximate_size()` function with `4000` for the argument named `size` and `False` for the argument named `a_kilobyte_is_1024_bytes`. (These named arguments happen to be in the same order as the arguments are listed in the function declaration, but that doesn't matter either.)

Program 3

```
from humansize import approximate_size
print (approximate_size(a_kilobyte_is_1024_bytes=False, size=4000))  #③
#'4.0 KB'
```

③ This calls the `approximate_size()` function with `False` for the argument named `a_kilobyte_is_1024_bytes` and `4000` for the argument named `size`. (See? I told you the order didn't matter.)

Program 4

```
from humansize import approximate_size

print (approximate_size(a_kilobyte_is_1024_bytes=False, 4000))       #④
# File "/usercode/__ed_file.py", line 11
# print (approximate_size(a_kilobyte_is_1024_bytes=False, 4000)) #\u2463
# ^
#SyntaxError: non-keyword arg after keyword arg
```

④ This call fails, because you have a named argument followed by an unnamed (positional) argument, and that never works. Reading the argument list from left to right, once you have a single named argument, the rest of the arguments must also be named.

Program 5

```
from humansize import approximate_size

print (approximate_size(size=4000, False) )                          #⑤
# File "/usercode/__ed_file.py", line 3
# print (approximate_size(size=4000, False) )
# ^
#SyntaxError: non-keyword arg after keyword arg
```

⑤ This call fails too, for the same reason as the previous call. Is that surprising? After all, you passed `4000` for the argument named `size`, then "obviously" that `False` value was meant for the `a_kilobyte_is_1024_bytes` argument. But Python doesn't work that way. As soon as you have a named

argument, all arguments to the right of that need to be named arguments, too.