Chapter 4

Advanced Functions

In this chapter, we go beyond the basics of using functions. I'll assume you can define and work with functions taking default arguments:

```
>>> def foo(a, b, x=3, y=2):
...     return (a+b)/(x+y)
...
>>> foo(5, 0)
1.0
>>> foo(10, 2, y=3)
2.0
>>> foo(b=4, x=8, a=1)
0.5
```

Notice the last way `foo` is called: with the arguments out of order, and everything specified by key-value pairs. Not everyone knows that you can call *any* function in Python this way. So long as the value of each argument is unambiguously specified, Python doesn't care how you call the function (and this case, we specify `b`, `x` and `a` out of order, letting `y` be its default value). We'll leverage this flexibility later.

This chapter's topics are useful and valuable on their own. And they are important building blocks for some *extremely* powerful patterns, which you learn in later chapters. Let's get started!

4.1 Accepting & Passing Variable Arguments

The `foo` function above can be called with either 2, 3, or 4 arguments. Sometimes you want to define a function that can take *any* number of arguments - zero or more, in other words. In Python, it looks like this:

```
# Note the asterisk. That's the magic part
def takes_any_args(*args):
    print("Type of args: " + str(type(args)))
    print("Value of args: " + str(args))
```

See carefully the syntax here. `takes_any_args` is just like a regular function, except you put an asterisk right before the argument `args`. Within the function, `args` is a tuple:

```
>>> takes_any_args("x", "y", "z")
Type of args: <class 'tuple'>
Value of args: ('x', 'y', 'z')
>>> takes_any_args(1)
Type of args: <class 'tuple'>
Value of args: (1,)
>>> takes_any_args()
Type of args: <class 'tuple'>
Value of args: ()
>>> takes_any_args(5, 4, 3, 2, 1)
Type of args: <class 'tuple'>
Value of args: (5, 4, 3, 2, 1)
>>> takes_any_args(["first", "list"], ["another","list"])
Type of args: <class 'tuple'>
Value of args: (['first', 'list'], ['another', 'list'])
```

If you call the function with no arguments, `args` is an empty tuple. Otherwise, it is a tuple composed of those arguments passed, in order. This is different from declaring a function that takes a single argument, which happens to be of type list or tuple:

```
>>> def takes_a_list(items):
...     print("Type of items: " + str(type(items)))
...     print("Value of items: " + str(items))
...
>>> takes_a_list(["x", "y", "z"])
Type of items: <class 'list'>
Value of items: ['x', 'y', 'z']
>>> takes_any_args(["x", "y", "z"])
Type of args: <class 'tuple'>
Value of args: (['x', 'y', 'z'],)
```

In these calls to `takes_a_list` and `takes_any_args`, the argument `items` is a list of strings. We're calling both functions the exact same way, but what happens in each function is different. Within `takes_any_args`, the tuple named `args` has one element - and that element is the list `["x", "y", "z"]`. But in `takes_a_list`, `items` is the list itself.

This *args idiom gives you some *very* helpful programming patterns. You can work with arguments as an abstract sequence, while providing a potentially more natural interface for whomever calls the function.

Above, I've always named the argument args in the function signature. Writing *args is a well-followed convention, but you can choose a different name - the asterisk is what makes it a variable argument. For instance, this takes paths of several files as arguments:

```
def read_files(*paths):
    data = ""
    for path in paths:
        with open(path) as handle:
            data += handle.read()
    return data
```

Most Python programmers use *args unless there is a reason to name it something else.[1] That reason is usually readability; read_files is a good example. If naming it something other than args makes the code more understandable, do it.

Argument Unpacking

The star modifier works in the other direction too. Intriguingly, you can use it with *any* function. For example, suppose a library provides this function:

```
def order_book(title, author, isbn):
    """
    Place an order for a book.
    """
    print("Ordering '{}' by {} ({})".format(
        title, author, isbn))
    # ...
```

Notice there's no asterisk. Suppose in another, completely different library, you fetch the book info from this function:

```
def get_required_textbook(class_id):
    """
    Returns a tuple (title, author, ISBN)
    """
    # ...
```

[1]This seems to be deeply ingrained; once I abbreviated it *a, only to have my code reviewer demand I change it to *args. They wouldn't approve it until I changed it, so I did.

Again, no asterisk. Now, one way you can bridge these two functions is to store the tuple result from `get_required_textbook`, then unpack it element by element:

```
>>> book_info = get_required_textbook(4242)
>>> order_book(book_info[0], book_info[1], book_info[2])
Ordering 'Writing Great Code' by Randall Hyde (1593270038)
```

Writing code this way is tedious and error-prone; not ideal.

Fortunately, Python provides a better way. Let's look at a different function:

```
def normal_function(a, b, c):
    print("a: {} b: {} c: {}".format(a,b,c))
```

No trick here - it really is a normal, boring function, taking three arguments. If we have those three arguments as a list or tuple, Python can automatically "unpack" them for us. We just need to pass in that collection, prefixed with an asterisk:

```
>>> numbers = (7, 5, 3)
>>> normal_function(*numbers)
a: 7 b: 5 c: 3
```

Again, `normal_function` is just a regular function. We did not use an asterisk on the `def` line. But when we call it, we take a tuple called `numbers`, and pass it in with the asterisk in front. This is then unpacked *within the function* to the arguments `a`, `b`, and `c`.

There is a duality here. We can use the asterisk syntax both in *defining* a function, and in *calling* a function. The syntax looks very similar. But realize they are doing two different things. One is packing arguments into a tuple automatically - called "variable arguments"; the other is *un*-packing them - called "argument unpacking". Be clear on the distinction between the two in your mind.

Armed with this complete understanding, we can bridge the two book functions in a much better way:

```
>>> book_info = get_required_textbook(4242)
>>> order_book(*book_info)
Ordering 'Writing Great Code' by Randall Hyde (1593270038)
```

This is more concise (less tedious to type), and more maintainable. As you get used to the concepts, you'll find it increasingly natural and easy to use in the code you write.

Variable Keyword Arguments

So far we have just looked at functions with *positional* arguments - the kind where you declare a function like `def foo(a, b):`, and then invoke it like `foo(7, 2)`. You know that a=7 and b=2 within the function, because of the order of the arguments. Of course, Python also has keyword arguments:

```
>>> def get_rental_cars(size, doors=4,
...         transmission='automatic'):
...     template = "Looking for a {}-door {} car with {} transmission
   ...."
...     print(template.format(doors, size, transmission))
...
>>> get_rental_cars("economy", transmission='manual')
Looking for a 4-door economy car with manual transmission....
```

And remember, Python lets you call *any* function just using keyword arguments:

```
>>> def bar(x, y, z):
...     return x + y * z
...
>>> bar(z=2, y=3, x=4)
10
```

These keyword arguments won't be captured by the `*args` idiom. Instead, Python provides a different syntax - using two asterisks instead of one:

```
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print("{} -> {}".format(key, value))
```

The variable `kwargs` is a *dictionary*. (In contrast to `args` - remember, that was a tuple.) It's just a regular `dict`, so we can iterate through its key-value pairs with `.items()`:

```
>>> print_kwargs(hero="Homer", antihero="Bart",
...     genius="Lisa")
hero -> Homer
antihero -> Bart
genius -> Lisa
```

The arguments to `print_kwargs` are key-value pairs. This is regular Python syntax for calling functions; what's interesting is happening *inside* the function. There, a variable called `kwargs` is defined. It's a Python dictionary, consisting of the key-value pairs passed in when the function was called.

Here's another example, which has a regular positional argument, followed by arbitrary key-value pairs:

```
def set_config_defaults(config, **kwargs):
    for key, value in kwargs.items():
        # Do not overwrite existing values.
        if key not in config:
            config[key] = value
```

This is perfectly valid. You can define a function that takes some normal arguments, followed by zero or more key-value pairs:

```
>>> config = {"verbosity": 3, "theme": "Blue Steel"}
>>> set_config_defaults(config, bass=11, verbosity=2)
>>> config
{'verbosity': 3, 'theme': 'Blue Steel', 'bass': 11}
```

Like with `*args`, naming this variable `kwargs` is just a strong convention; you can choose a different name if that improves readability.

Keyword Unpacking

Just like with `*args`, double-star works the other way too. We can take a regular function, and pass it a dictionary using two asterisks:

```
>>> def normal_function(a, b, c):
...     print("a: {} b: {} c: {}".format(a,b,c))
...
>>> numbers = {"a": 7, "b": 5, "c": 3}
>>> normal_function(**numbers)
a: 7 b: 5 c: 3
```

Note the keys of the dictionary *must* match up with how the function was declared. Otherwise you get an error:

```
>>> bad_numbers = {"a": 7, "b": 5, "z": 3}
>>> normal_function(**bad_numbers)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: normal_function() got an unexpected keyword argument 'z'
```

This is called *keyword argument unpacking*. It works regardless of whether that function has default values for some of its arguments or not. So long as the value of each argument is specified one way or another, you have valid code:

```
>>> def another_function(x, y, z=2):
...     print("x: {} y: {} z: {}".format(x,y,z))
...
>>> all_numbers = {"x": 2, "y": 7, "z": 10}
>>> some_numbers = {"x": 2, "y": 7}
>>> missing_numbers = {"x": 2}
>>> another_function(**all_numbers)
x: 2 y: 7 z: 10
>>> another_function(**some_numbers)
x: 2 y: 7 z: 2
>>> another_function(**missing_numbers)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: another_function() missing 1 required positional argument: '
   y'
```

Combining Positional and Keyword Arguments

You can combine the syntax to use both positional and keyword arguments. In a function signature, just separate `*args` and `**kwargs` by a comma:

```
>>> def general_function(*args, **kwargs):
...     for arg in args:
...         print(arg)
...     for key, value in kwargs.items():
...         print("{} -> {}".format(key, value))
...
>>> general_function("foo", "bar", x=7, y=33)
foo
bar
y -> 33
x -> 7
```

This usage - declaring a function like `def general_function(*args, **kwargs)` - is the most general way to define a function in Python. A function so declared can be called in any way, with any valid combination of keyword and non-keyword arguments - including no arguments.

Similarly, you can call a function using both - and both will be unpacked:

```
>>> def addup(a, b, c=1, d=2, e=3):
...     return a + b + c + d + e
...
>>> nums = (3, 4)
>>> extras = {"d": 5, "e": 2}
>>> addup(*nums, **extras)
15
```

There's one last point to understand, on argument ordering. When you `def` the function, you specify the arguments in this order:

- Named, regular (non-keyword) arguments, then
- the `*args` non-keyword variable arguments, then
- the `**kwargs` keyword variable arguments, and finally
- required keyword-only arguments.

You can omit any of these when defining a function. But any that are present *must* be in this order.

```
# All these are valid function definitions.
def combined1(a, b, *args): pass
def combined2(x, y, z, **kwargs): pass
def combined3(*args, **kwargs): pass
def combined4(x, *args): pass
def combined5(u, v, w, *args, **kwargs): pass
def combined6(*args, x, y): pass
```

Violating this order will cause errors:

```
>>> def bad_combo(**kwargs, *args): pass
  File "<stdin>", line 1
    def bad_combo(**kwargs, *args): pass
                          ^
SyntaxError: invalid syntax
```

Sometimes you might want to define a function that takes 0 or more positional arguments, and 1 or more *required* keyword arguments. You can define a function like this with `*args` followed by regular arguments, forming a special category, called *keyword-only arguments*. If present, whenever that function is called, all must specified as key-value pairs, *after* the non-keyword arguments:

```
>>> def read_data_from_files(*paths, format):
...     """Read and merge data from several files,
...     which are in XML, JSON, or YAML format."""
...     # ...
...
>>> housing_files = ["houses.json", "condos.json"]
>>> housing_data = read_data_from_files(
...     *housing_files, format="json")
>>> commodities_data = read_data_from_files(
        "commodities.xml", format="xml")
```

See how format's value is specified with a key-value pair. If you try passing it without format= in front, you get an error:

```
>>> commodities_data = read_data_from_files(
...     "commodities.xml", "xml")
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: read_data_from_files() missing 1 required keyword-only
   argument: 'format'
```

4.2 Functions As Objects

In Python, functions are ordinary objects - just like an integer, a list, or an instance of a class you create. The implications are profound, letting you do certain *very* useful things with functions. Leveraging this is one of those secrets separating average Python developers from great ones, because of the *extremely* powerful abstractions which follow.

Once you get this, it can change the way you write software forever. In fact, these advanced patterns for using functions in Python largely transfer to other languages you will use in the future.

To explain, let's start by laying out a problematic situation, and how to solve it. Imagine you have a list of strings representing numbers:

```
nums = ["12", "7", "30", "14", "3"]
```

Suppose we want to find the biggest integer in this list. The max builtin does not help us:

```
>>> max(nums)
'7'
```

This isn't a bug, of course; since the objects in nums are strings, max compares each element lexicographically.[2] By that criteria, "7" is greater than "30", for the same reason "g" comes after

[2]Meaning, alphabetically, but generalizing beyond the letters of the alphabet.

"ca" alphabetically. Essentially, max is evaluating the element by a different criteria than what we want.

Since max's algorithm is simple, let's roll our own that compares based on the integer value of the string:

```
>>> def max_by_int_value(items):
...     # For simplicity, assume len(items) > 0
...     biggest = items[0]
...     for item in items[1:]:
...         if int(item) > int(biggest):
...             biggest = item
...     return biggest
...
>>> max_by_int_value(nums)
'30'
```

This gives us what we want: it returns the element in the original list which is maximal, as evaluated by our criteria. Now imagine working with different data, where you have different criteria. For example, a list of actual integers:

```
integers = [3, -2, 7, -1, -20]
```

Suppose we want to find the number with the greatest *absolute value* - i.e., distance from zero. That would be -20 here, but standard max won't do that:

```
>>> max(integers)
7
```

Again, let's roll our own, using the built-in abs function:

```
>>> def max_by_abs(items):
...     biggest = items[0]
...     for item in items[1:]:
...         if abs(item) > abs(biggest):
...             biggest = item
...     return biggest
...
>>> max_by_abs(integers)
-20
```

One more example - a list of dictionary objects:

```
student_joe = {'gpa': 3.7, 'major': 'physics',
               'name': 'Joe Smith'}
student_jane = {'gpa': 3.8, 'major': 'chemistry',
                'name': 'Jane Jones'}
student_zoe = {'gpa': 3.4, 'major': 'literature',
               'name': 'Zoe Fox'}
students = [student_joe, student_jane, student_zoe]
```

Now, what if we want the record of the student with the highest GPA? Here's a suitable max function:

```
>>> def max_by_gpa(items):
...     biggest = items[0]
...     for item in items[1:]:
...         if item["gpa"] > biggest["gpa"]:
...             biggest = item
...     return biggest
...
>>> max_by_gpa(students)
{'name': 'Jane Jones', 'gpa': 3.8, 'major': 'chemistry'}
```

Just one line of code is different between `max_by_int_value`, `max_by_abs`, and `max_by_gpa`: the comparison line. `max_by_int_value` says `if int(item) > int(biggest)`; `max_by_abs` says `if abs(item) > abs(biggest)`; and `max_by_gpa` compares `item["gpa"]` to `biggest["gpa"]`. Other than that, these `max` functions are identical.

I don't know about you, but having nearly-identical functions like this drives me nuts. The way out is to realize the comparison is based on a value *derived* from the element - not the value of the element itself. In other words: each cycle through the for loop, the two elements are **not** themselves compared. What is compared is some derived, calculated value: `int(item)`, or `abs(item)`, or `item["gpa"]`.

It turns out we can abstract out that calculation, using what we'll call a *key function*. A key function is a function that takes exactly one argument - an element in the list. It returns the derived value used in the comparison. In fact, `int` works like a function, even though it's technically a type, because `int("42")` returns 42.[3] So types and other callables work, as long as we can invoke it like a one-argument function.

This lets us define a very generic max function:

[3]Python uses the word *callable* to describe something that can be invoked like a function. This can be an actual function, a type or class name, or an object defining the `__call__` magic method. Key functions are frequently actual functions, but can be any callable.

```
>>> def max_by_key(items, key):
...     biggest = items[0]
...     for item in items[1:]:
...         if key(item) > key(biggest):
...             biggest = item
...     return biggest
...
>>> # Old way:
... max_by_int_value(nums)
'30'
>>> # New way:
... max_by_key(nums, int)
'30'
>>> # Old way:
... max_by_abs(integers)
-20
>>> # New way:
... max_by_key(integers, abs)
-20
```

Pay attention: you are passing the function object itself - `int` and `abs`. You are *not* invoking the key function in any direct way. In other words, you write `int`, not `int()`. This function object is then called as needed by `max_by_key`, to calculate the derived value:

```
        # key is actually int, abs, etc.
        if key(item) > key(biggest):
```

For sorting the students by GPA, we need a function extracting the "gpa" key from each student dictionary. There is no built-in function that does this, but we can define our own and pass it in:

```
>>> # Old way:
... max_by_gpa(students)
{'gpa': 3.8, 'name': 'Jane Jones', 'major': 'chemistry'}

>>> # New way:
... def get_gpa(who):
...     return who["gpa"]
...
>>> max_by_key(students, get_gpa)
{'gpa': 3.8, 'name': 'Jane Jones', 'major': 'chemistry'}
```

Again, notice `get_gpa` is a function object, and we are passing that function itself to `max_by_key`. We never invoke `get_gpa` directly; `max_by_key` does that automatically.

You may be realizing now just how powerful this can be. In Python, functions are simply objects - just as much as an integer, or a string, or an instance of a class is an object. You can store functions in variables; pass them as arguments to other functions; and even return them from other function and method calls. This all provides new ways for you to encapsulate and control the behavior of your code.

The Python standard library demonstrates some excellent ways to use such functional patterns. Let's look at a key (ha!) example.

4.3 Key Functions in Python

Earlier, we saw the built-in `max` doesn't magically do what we want when sorting a list of numbers-as-strings:

```
>>> nums = ["12", "7", "30", "14", "3"]
>>> max(nums)
'7'
```

Again, this isn't a bug - `max` just compares elements according to the data type, and `"7" > "12"` evaluates to `True`. But it turns out `max` is customizable. You can pass it a key function!

```
>>> max(nums, key=int)
'30'
```

The value of `key` is a function taking one argument - an element in the list - and returning a value for comparison. But `max` isn't the only built-in accepting a key function. `min` and `sorted` do as well:

```
>>> # Default behavior...
... min(nums)
'12'
>>> sorted(nums)
['12', '14', '3', '30', '7']
>>>
>>> # And with a key function:
... min(nums, key=int)
'3'
>>> sorted(nums, key=int)
['3', '7', '12', '14', '30']
```

Many algorithms can be cleanly expressed using `min`, `max`, or `sorted`, along with an appropriate key function. Sometimes a built-in (like `int` or `abs`) will provide what you need, but often you'll want to create a custom function. Since this is so commonly needed, the `operator` module provides some helpers. Let's revisit the example of a list of student records.

```
>>> student_joe = {'gpa': 3.7, 'major': 'physics',
        'name': 'Joe Smith'}
>>> student_jane = {'gpa': 3.8, 'major': 'chemistry',
        'name': 'Jane Jones'}
>>> student_zoe = {'gpa': 3.4, 'major': 'literature',
        'name': 'Zoe Fox'}
>>> students = [student_joe, student_jane, student_zoe]
>>>
>>> def get_gpa(who):
...     return who["gpa"]
...
>>> sorted(students, key=get_gpa)
[{'gpa': 3.4, 'major': 'literature', 'name': 'Zoe Fox'},
 {'gpa': 3.7, 'major': 'physics', 'name': 'Joe Smith'},
 {'gpa': 3.8, 'major': 'chemistry', 'name': 'Jane Jones'}]
```

This is effective, and a fine way to solve the problem. Alternatively, the `operator` module's `itemgetter` creates and returns a key function that looks up a named dictionary field:

```
>>> from operator import itemgetter
>>>
>>> # Sort by GPA...
... sorted(students, key=itemgetter("gpa"))
[{'gpa': 3.4, 'major': 'literature', 'name': 'Zoe Fox'},
 {'gpa': 3.7, 'major': 'physics', 'name': 'Joe Smith'},
 {'gpa': 3.8, 'major': 'chemistry', 'name': 'Jane Jones'}]
>>>
>>> # Now sort by major:
... sorted(students, key=itemgetter("major"))
[{'gpa': 3.8, 'major': 'chemistry', 'name': 'Jane Jones'},
 {'gpa': 3.4, 'major': 'literature', 'name': 'Zoe Fox'},
 {'gpa': 3.7, 'major': 'physics', 'name': 'Joe Smith'}]
```

Notice `itemgetter` is a function that creates and returns a function - itself a good example of how to work with function objects. In other words, the following two key functions are completely equivalent:

```
# What we did above:
def get_gpa(who):
    return who["gpa"]

# Using itemgetter instead:
from operator import itemgetter
get_gpa = itemgetter("gpa")
```

This is how you use `itemgetter` when the sequence elements are dictionaries. It also works when the elements are tuples or lists - just pass a number index instead:

```
>>> # Same data, but as a list of tuples.
... student_rows = [
...       ("Joe Smith", "physics", 3.7),
...       ("Jane Jones", "chemistry", 3.8),
...       ("Zoe Fox", "literature", 3.4),
...       ]
>>>
>>> # GPA is the 3rd item in the tuple, i.e. index 2.
... # Highest GPA:
... max(student_rows, key=itemgetter(2))
('Jane Jones', 'chemistry', 3.8)
>>>
>>> # Sort by major:
... sorted(student_rows, key=itemgetter(1))
[('Jane Jones', 'chemistry', 3.8),
 ('Zoe Fox', 'literature', 3.4),
 ('Joe Smith', 'physics', 3.7)]
```

`operator` also provides `attrgetter`, for keying off an attribute of the element, and `methodcaller` for keying off a method's return value - useful when the sequence elements are instances of your own class:

```
>>> class Student:
...     def __init__(self, name, major, gpa):
...         self.name = name
...         self.major = major
...         self.gpa = gpa
...     def __repr__(self):
...         return "{}: {}".format(self.name, self.gpa)
...
>>> student_objs = [
...      Student("Joe Smith", "physics", 3.7),
...      Student("Jane Jones", "chemistry", 3.8),
...      Student("Zoe Fox", "literature", 3.4),
...      ]
>>> from operator import attrgetter
>>> sorted(student_objs, key=attrgetter("gpa"))
[Zoe Fox: 3.4, Joe Smith: 3.7, Jane Jones: 3.8]
```