* **Variables**, which store values for later use
* **Data types**, such as numbers and booleans
* **Whitespace**, which separates statements
* **Comments**, which make your code easier to read
* **Arithmetic operations**, including +, -, \*, /, \*\*, and %

IndentationError: expected an indented block

You'll get this error whenever your whitespace is off.

On Codecademy, we will use two-space indentation (two blank spaces for each indentation) to make sure you can easily read code with multiple indentations in your browser. Beyond Codecademy, you will also see Python code that uses four-space indentation.

The # sign is for comments. A comment is a line of text that Python won't try to run as code. It's just for humans to read.

Instead, for multi-line comments, you can include the whole block in a set of triple quotation marks:

"""Sipping from your cup 'til it runneth over, Holy Grail. """

We create a new variable called eight and set it to 8, or the result of 2 to the power to 3 (2^3).

eight = 2 \*\* 3

In the above example, we create a new variable called eight and set it to 8, or the result of 2 to the power to 3 (2^3).

Notice that we use \*\* instead of \* or the multiplication operator.

You'll have to divide 6.75 by 100 in order to get the decimal form of the percentage.

Three ways to create strings

'Alpha'

"Bravo"

str(3)

String methods

len("Charlie")

"Delta".upper()

"Echo".lower()

Printing a string

print "Foxtrot"

len():print len(parrot)

Methods that use dot notation only work with strings.

On the other hand, len() and str() can work on other data types.

print "Life " + "of " + "Brian"

This will print out the phrase Life of Brian.

Escaping characters

There are some characters that cause problems. For example:

'There's a snake in my boot!'

This code breaks because Python thinks the apostrophe in 'There's' ends the string. We can use the backslash to fix the problem, like this:

'There\'s a snake in my boot!

Each character in a string is assigned a number. This number is called the **index**. Check out the diagram in the editor.

c = "cats"[0]

n = "Ryan"[3]

The % operator after a string is used to combine a string with variables. The % operator will replace a %sin the string with the string variable that comes after it.

name = "Mike"

print "Hello %s" % (name)

You need the same number of %s terms in a string as the number of variables in parentheses

string\_1 = "Camelot"

string\_2 = "place"

print "Let's not go to %s. 'This a silly %s." % (string\_1, string\_2)

We can use a function called datetime.now() to retrieve the current date and time. Notice how the output looks like 2016-11-25 23:45:14.317454.

Let's combine the two!

from datetime import datetime

now = datetime.now()

print '%s/%s/%s' % (now.month, now.day, now.year)

print '%s:%s:%s' % (now.hour, now.minute, now.second)

Print today's date in the following format? mm/dd/yyyy.

print the current time in the pretty form of hh:mm:ss

Remember that the % operator will fill the %s placeholders in the string on the left with the strings in the parentheses on the right.

**Control flow** gives us this ability to choose among outcomes based off what else is happening in the program.

here's an order of operations for boolean operators:

1. not is evaluated first;
2. and is evaluated next;
3. or is evaluated last.

if 8 > 9:

print "I don't get printed!"

elif 8 < 9:

print "I get printed!"

else:

print "I also don't get printed!"

name = raw\_input("What's your name?")

print name

raw\_input() accepts a string, prints it, and then waits for the user to type something and press Enter (or Return).

x = "J123"

x.isalpha() # False

In the first line, we create a string with letters and numbers.

The second line then runs the method .isalpha()which returns False since the string contains non-letter characters.

s = "Charlie"

print s[1:4] # will print "har"

Then we access a slice of "Charlie" using s[1:4]. This returns everything from the letter at position [1, 4-1]

Here's the full function pieced together:

def hello\_world():

"""Prints 'Hello World!' to the console."""

print "Hello World!"

1. The *header*, which includes the def keyword, the name of the function, and any *parameters* the function requires.
2. An optional *comment* that explains what the function does.
3. The *body*, which describes the procedures the function carries out. The body is *indented*, just like conditional statements.

A parameter acts as a variable name for a passed in **argument**.

There is a Python module named math that includes a number of useful variables and functions. In order to access math, all you need is the import keyword. When you simply import a module this way, it's called a **generic import**.

import math

print math.sqrt(25)

Pulling in just a single function from a module is called a function import, and it's done with the from keyword:

from module import function

from math import sqrt

print sqrt(25)

want all of the variables and functions in a module, Universal import can handle this for you.

from module import \*

from math import \*

print sqrt(25)

it's best to stick with either import module and type module.name or just import specific variables and functions from various modules as needed.

some of the functions that are built in to Python (no modules required!)

max(1,2,3) will return 3 (the largest number in the set of arguments).

min() then returns the smallest of a given series of arguments.

The abs() function returns the **absolute value** of the number it takes as an argument

the type() function returns the type of the data it receives as an argument.

print type(42)

print type(4.2)

print type('spam')

Python will output:

<type 'int'>

<type 'float'>

<type 'str'>

Lists are a datatype you can use to store a collection of different pieces of information as a sequence under a single variable name.

list\_name = [item\_1, item\_2]

A list can also be empty: empty\_list = [].

The index appears directly after the list name, in between brackets, like this: list\_name[index].

letters = ['a', 'b', 'c']

letters.append('d')

print len(letters)

print letters

Sometimes, you only want to access a portion of a list.

letters = ['a', 'b', 'c', 'd', 'e']

slice = letters[1:3] //print ‘b’, ‘c’

If your list slice includes the very first or last item in a list (or a string), the index for that item doesn't have to be included.

my\_list[:2] # Grabs the first two items

my\_list[3:] # Grabs the fourth through last items

print the first index that contains the string "bat", which will print 1.

animals = ["ant", "bat", "cat"] print animals.index("bat")

We can also insert items into a list.

animals.insert(1, "dog") print animals

If you want to do something with every item in the list, you can use a for loop.

for variable in list\_name: # Do stuff!

If your list is a jumbled mess, you may need to sort()it.

animals = ["cat", "ant", "bat"]

animals.sort()

A dictionary is similar to a list, but you access values by looking up a *key* instead of an index. A key can be any string or number. Dictionaries are enclosed in curly braces, like so:

d = {'key1' : 1, 'key2' : 2, 'key3' : 3}

Accessing dictionary values by key is just like accessing list values by index:

residents['Puffin']# Gets the value 104

add new key/value pairs to the dictionary after it is created like so:

dict\_name[new\_key] = new\_value

An empty pair of curly braces {} is an empty dictionary, just like an empty pair of [] is an empty list.

The length len() of a dictionary is the number of key-value pairs it has. Each pair counts only once, even if the value is a list. (That's right: you can put lists *inside* dictionaries!)

Items can be removed from a dictionary with the del command:

del dict\_name[key\_name]

will remove the key key\_name and its associated value from the dictionary.

A new value can be associated with a key by assigning a value to the key, like so:

dict\_name[key] = new\_value

Sometimes you need to remove something from a list.

beatles = ["john","paul","george","ringo","stuart"]

beatles.remove("stuart")

The key "fish" has a *list*, the key "cash" has an *int*, and the key "luck" has a *string*.

my\_dict = {

"fish": ["c", "a", "r", "p"],

"cash": -4483,

"luck": "good" }

print my\_dict["fish"][0]

So running the following:

for item in [1, 3, 21]:

print item

would print 1, then 3, and then 21.

You can also use a for loop on a dictionary to loop through its keys with the following:

d = {"foo" : "bar"}

for key in d:

print d[key] # prints "bar"

Note that dictionaries are unordered, meaning that any time you loop through a dictionary, you will go through every key, and print every value, but you are not guaranteed to get them in any particular order.

for letter in "Codecademy":

print letter

Now that you have all of your product info, you should print out all of your inventory information.

once = {'a': 1, 'b': 2}

twice = {'a': 2, 'b': 4}

for key in once:

print "Once: %s" % once[key]

print "Twice: %s" % twice[key]

1. In the above example, we create two dictionaries, once and twice, that have the same keys.
2. Because we know that they have the same keys, we can loop through one dictionary and print values from both once and twice.

float(5) / 2 # 2.5

To divide two integers and end up with a float, you must first use float() to convert one of the integers to a float.

return 0.9 \* average(cost["apples"]) + \

0.1 \* average(cost["bananas"])

The \ character is a *continuation character*. The following line is considered a *continuation* of the current line.

.append() elements to the end of a list.

For a list called n:

1. n.pop(index) will remove the item at indexfrom the list and return it to you:

n = [1, 3, 5] n.pop(1) # Returns 3 (the item at index 1) print n # prints [1, 5]

1. n.remove(item) will remove the actual item if it finds it:

n.remove(1) # Removes 1 from the list, # NOT the item at index 1 print n # prints [3, 5]

1. del(n[1]) is like .pop in that it will remove the item at the given index, but it won't return it:

del(n[1]) # Doesn't return anything print n # prints [1, 5]

using a list as an argument in a function is essentially the same as using just a number or string!

The Python range() function is just a shortcut for generating a list, so you can use ranges in all the same places you can use lists.

range(6) # => [0, 1, 2, 3, 4, 5] range(1, 6) # => [1, 2, 3, 4, 5] range(1, 6, 3) # => [1, 4]

The range function has three different versions:

1. range(stop)
2. range(start, stop)
3. range(start, stop, step)

We have two ways of iterating through a list.

**Method 1** - for item in list:

for item in list: print item

**Method 2** - iterate through indexes:

for i in range(len(list)): print list[i]

Method 1 is useful to loop through the list, but it's not possible to modify the list this way.

Method 2 uses indexes to loop through the list, making it possible to also modify the list if needed.

Using multiple lists in a function is no different from just using multiple arguments in a function!

a = [1, 2, 3] b = [4, 5, 6]

print a + b # prints [1, 2, 3, 4, 5, 6]

The example above is just a reminder of how to concatenate two lists.

Finally, this exercise shows how to make use of a single list that contains multiple lists and how to use them in a function.

list\_of\_lists = [[1, 2, 3], [4, 5, 6]]

for lst in list\_of\_lists:

for item in lst:

print item

we'll make into a 5 x 5 grid of all "O"s, for "ocean."

print ["O"] \* 5

will print out ['O', 'O', 'O', 'O', 'O'], which is the basis for a row of our board.

We'll do this five times to make five rows.

letters = ['a', 'b', 'c', 'd']

print " ".join(letters)

print "---".join(letters)

1. In the example above, we create a list called letters.
2. Then, we print a b c d. The .join method uses the string to combine the items in the list.
3. Finally, we print a---b---c---d. We are calling the .join function on the "---" string.

from random import randint

coin = randint(0, 1)

dice = randint(1, 6)

1. In the above example, we first import the randint(low, high) function from the random module.
2. Then, we generate either zero or one and store it in coin.
3. Finally, we generate a number from one to six inclusive.

number = raw\_input("Enter a number: ")

f int(number) == 0: print "You entered 0"

raw\_input asks the user for input and returns it as a string. But we're going to want to use integers for our guesses! To do this, we'll wrap the raw\_inputs with int() to convert the string to an integer.

The while loop is similar to an if statement: it executes the code inside of it if some condition is true. The difference is that the while loop will continue to execute as long as the condition is true.

An infinite loop is a loop that never exits. This can happen for a few reasons:

1. The loop condition cannot possibly be false (e.g. while 1 != 2)
2. The logic of the loop prevents the loop condition from becoming false.

count = 10

while count > 0:

count += 1 # Instead of count -= 1

The break is a one-line statement that means "exit the current loop."

while/else is similar to if/else, but there is a difference: the else block will execute **anytime** the loop condition is evaluated to False. This means that it will execute if the loop is never entered or if the loop exits normally. If the loop exits as the result of a break, the else will not be executed.

guess = int(raw\_input("Your guess: "))

Remember, raw\_input turns user input into a string, so we use int() to make it a number again.

An alternative way to loop is the for loop. This example means "for each number i in the range 0 - 9, print i".

for i in range(10):

print i

word = "Marble" for char in word: print char,

The example above iterates through each character in word and, in the end, prints out M a r b l e.

The , character after our print statement means that our next print statement keeps printing on the same line.

You may be wondering how looping over a dictionary would work.

The short answer is: you get the key which you can use to get the value.

d = {'x': 9, 'y': 10, 'z': 20}

for key in d:

if d[key] == 10:

print "This dictionary has the value 10!"

A weakness of using this for-each style of iteration is that you don't know the index of the thing you're looking at. Thankfully the built-in enumerate function helps with this.

enumerate works by supplying a corresponding index to each element in the list that you pass it. Each time you go through the loop, index will be one greater, and item will be the next item in the sequence. It's very similar to using a normal for loop with a list, except this gives us an easy way to count how many items we've seen so far.

choices = ['pizza', 'pasta', 'salad', 'nachos']

for index, item in enumerate(choices):

print index, item

It's also common to need to iterate over two lists at once. This is where the built-in zip function comes in handy.

zip will create pairs of elements when passed two lists, and will stop at the end of the shorter list.

zip can handle three or more lists as well!

list\_a = [3, 9, 17, 15, 19]

list\_b = [2, 4, 8, 10, 30, 40, 50, 60, 70, 80, 90]

for a, b in zip(list\_a, list\_b):

# Add your code here!

print max(a, b)

Just like with while, for loops may have an else associated with them.

In this case, the else statement is executed after the for, but only if the for ends normally—that is, not with a break

def is\_prime(x):

if x < 2:

return False

else:

for n in range(2, x-1):

if x % n == 0:

return False

return True

Write a function called censor that takes two strings, text and word, as input. It should return the text with the word you chose replaced with asterisks. For example:

censor("this hack is wack hack", "hack")

should return:

"this \*\*\*\* is wack \*\*\*\*"

* Assume your input strings won't contain punctuation or upper case letters.
* The number of asterisks you put should correspond to the number of letters in the censored word.

def censor(text, word):

words = text.split()

result = ''

stars = '\*' \* len(word)

count = 0

for i in words:

if i == word:

words[count] = stars

count += 1

result =' '.join(words) #add space between words

return result

Write a function remove\_duplicates that takes in a list and removes elements of the list that are the same.

For example: remove\_duplicates([1, 1, 2, 2]) should return [1, 2].

* Don't remove every occurrence, since you need to keep a single occurrence of a number.

def remove\_duplicates(nums):

res = []

if nums == []:

return res

# Sort the input list from low to high

nums = sorted(nums)

# Initialize the output list, give it the first value of the now-sorted input list

res.append(nums[0])

# Go through the values of the sorted list and append to the output list

# any values that are greater than the last value of the output list

for item in nums:

if item > res[len(res) - 1]:

res.append(item)

return res

Recall that a dictionary is just a collection of keys and values.

d = { "Name": "Guido", "Age": 56, "BDFL": True }

print d.items()

# => [('BDFL', True), ('Age', 56), ('Name', 'Guido')]

While .items() returns an array of *tuples* with each tuple consisting of a key/value pair from the dictionary:

* The .keys() method returns a list of the dictionary's keys, and
* The .values() method returns a list of the dictionary's values.

Again, these methods will not return the keys or values from the dictionary in any specific order.

You can use in very intuitively, like so:

for number in range(5):

print number,

d = { "name": "Eric", "age": 26 }

for key in d:

print key, d[key],

for letter in "Eric":

print letter, # note the comma!

1. In the example above, first we create and iterate through a range, printing out 0 1 2 3 4. Note that the trailing comma ensures that we keep printing on the same line.
2. Next, we create a dictionary and iterate through, printing out age 26 name Eric. Dictionaries have no specific order.
3. Finally, we iterate through the letters of a string, printing out E r i c.

List comprehension

If you want those numbers doubled, you could use:

doubles = [x \* 2 for x in range(1, 6)]

# => [2, 4, 6, 8, 10]

c = ['C' for x in range(5) if x < 3] print c

The example above creates and prints out a list containing ['C', 'C', 'C'].

List slicing allows us to access elements of a list in a concise manner. The syntax looks like this:

[start:end:stride]

Where start describes where the slice starts (inclusive), end is where it ends (exclusive), and stride describes the space between items in the sliced list. For example, a stride of 2 would select every other item from the original list to place in the sliced list.

If you don't pass a particular index to the list slice, Python will pick a default.

to\_five = ['A', 'B', 'C', 'D', 'E']

print to\_five[3:] # prints ['D', 'E']

print to\_five[:2] # prints ['A', 'B']

print to\_five[::2] # print ['A', 'C', 'E']

1. The default starting index is 0.
2. The default ending index is the end of the list.
3. The default stride is 1.

A negative stride progresses through the list from right to left to reverse the list.

letters = ['A', 'B', 'C', 'D', 'E'] print letters[::-1]

In the example above, we print out ['E', 'D', 'C', 'B', 'A'].

**functional programming**, which means that you're allowed to pass functions around just as if they were variables or values.

Typing

lambda x: x % 3 == 0

Is the same as

def by\_three(x): return x % 3 == 0

Only we don't need to actually give the function a name; it does its work and returns a value without one. That's why the function the lambda creates is an anonymous function.

Remember, filter() takes two arguments: the first is the function that tells it what to filter, and the second is the object to perform the filtering on.

my\_list = range(16) filter(lambda x: x % 3 == 0, my\_list)

Lambdas are useful when you need a quick function to do some work for you.

If you plan on creating a function you'll use over and over, you're better off using def and giving that function a name.

***Bitwise operations*** are operations that directly manipulate *bits*.

print 5 >> 4 # Right Shift

print 5 << 1 # Left Shift

print 8 & 5 # Bitwise AND, directly operate on integer

print 9 | 4 # Bitwise OR

print 12 ^ 42 # Bitwise XOR

print ~88 # Bitwise NOT

In Python, you can write numbers in binary format by starting the number with 0b.

In order to print a number in its binary representation, you can use the bin()function. bin() takes an integer as input and returns the binary representation of that integer in a string. (Keep in mind that after using the binfunction, you can no longer operate on the value like a number.)

You can also represent numbers in base 8 and base 16 using the oct() and hex() functions.

Python has an int() function that you've seen a bit of already. It can turn non-integer input into an integer, like this:

int("42") # ==> 42

When given a string containing a number and the base that number is in, the function will return the value of that number converted to base ten.

int("110", 2) # ==> 6

Note that you can only do bitwise operations on an **integer**. Trying to do them on strings or floats will result in nonsensical output!

Note that the bitwise | operator can only create results that are greater than or equal to the larger of the two integer inputs.

The bitwise NOT operator (~) just flips all of the bits in a single number.

A *bit mask* is just a variable that aids you with bitwise operations.

You can use masks to check a bit in a number on using &. Num&0b1000

You can also use masks to turn a bit in a number on using |. Num|0b1000

Using ^ on a bit with the number one will return a result where that bit is flipped.

Finally, you can also use the left shift (<<) and right shift (>>) operators to slide masks into place.

a = 0b101 # Tenth bit mask

mask = (0b1 << 9) # One less than ten

desired = a ^ mask

Let's say that I want to turn on the 10th bit from the right of the integer a.

Instead of writing out the entire number, we slide a bit over using the << operator.

**Class**

Defining a class is much like defining a function, but we use the class keyword instead. We also use the word object in parentheses because we want our classes to *inherit* the object class. This means that our class has all the properties of an object, which is the simplest, most basic class.

class NewClass(object): # Class magic here

This gives them the powers and abilities of a Python object. By convention, user-defined Python class names start with a capital letter.

\_\_init\_\_(). This function is required for classes, and it's used to **initialize** the objects it creates. \_\_init\_\_() always takes at least one argument, self, that refers to the object being created.

go ahead and **instantiate** (create) our first object.

The part that *is* magic is the fact that self is the *first* parameter passed to \_\_init\_\_(). Python will use the first parameter that \_\_init\_\_() receives to refer to the object being created; this is why it's often called self, since this parameter gives the object being created its identity.

We can access attributes of our objects using *dot notation*. Here's how it works:

class Square(object):

def \_\_init\_\_(self):

self.sides = 4

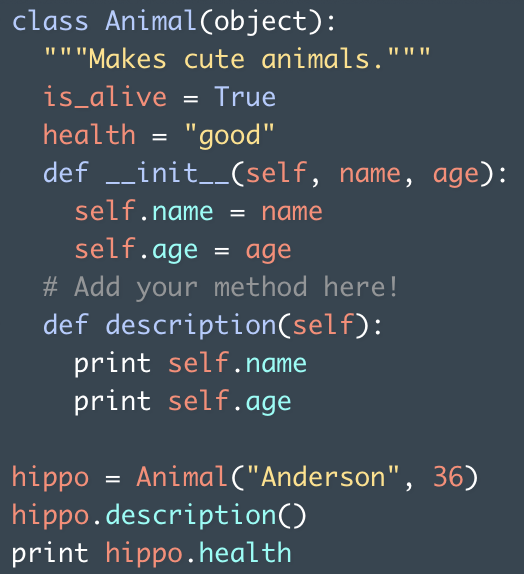
my\_shape = Square()

print my\_shape.sides

1. First we create a class named Square with an attribute sides.
2. Outside the class definition, we create a new instance of Square named my\_shape and access that attribute using my\_shape.sides.

Note that self is only used in the \_\_init\_\_() function definition; we don't need to pass it to our instance objects.

When dealing with classes, you can have variables that are available everywhere (*global variables*), variables that are only available to members of a certain class (*member variables*), and variables that are only available to particular instances of a class (*instance variables*).



Just like when we defined \_\_init\_\_(), you need to provide self as the first argument of any class method.

Inheritance is the process by which one class takes on the attributes and methods of another, and it's used to express an *is-a* relationship.

In Python, inheritance works like this:

class DerivedClass(BaseClass): # code goes here

where DerivedClass is the new class you're making and BaseClass is the class from which that new class inherits.

class Employee(object):

def \_\_init\_\_(self, name):

self.name = name

def greet(self, other):

print "Hello, %s" % other.name

class CEO(Employee):

def greet(self, other):

print "Get back to work, %s!" % other.name

ceo = CEO("Emily")

emp = Employee("Steve")

emp.greet(ceo) # Hello, Emily

ceo.greet(emp) # Get back to work, Steve!

Rather than have a separate greet\_underling method for our CEO, we override (or re-create) the greet method on top of the base Employee.greet method. This way, we don't need to know what type of Employee we have before we greet another Employee.

You can directly access the attributes or methods of a superclass with Python's built-in super call.

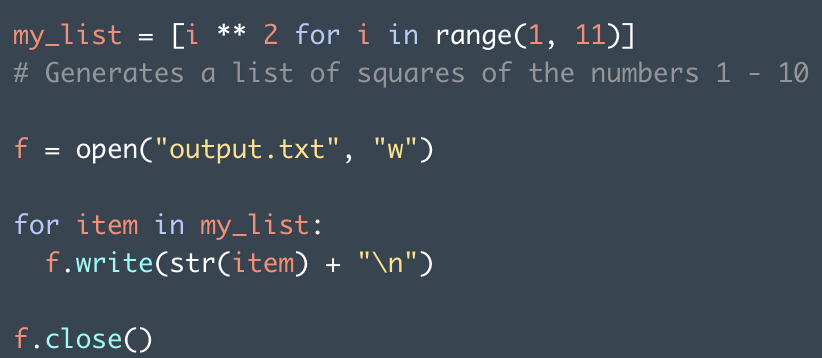
class Derived(Base):

def m(self):

return super(Derived, self).m()

Where m() is a method from the base class.

One useful class method to override is the built-in \_\_repr\_\_() method, which is short for *representation*; by providing a return value in this method, we can tell Python how to represent an object of our class (for instance, when using a print statement).



The first code that you saw executed in the previous exercise was this:

f = open("output.txt", "w")

This told Python to open output.txt in "w" mode ("w" stands for "write").

We stored the result of this operation in a file object, f.

Doing this opens the file in write-mode and prepares Python to send data into the file.

You can open files in any of the following modes:

* write-only mode ("w")
* read-only mode ("r")
* read and write mode ("r+")
* append mode ("a"), which adds any new data you write to the file to the end of the file.

The .write() method takes a string argument.

You must close the file. You do this simply by calling my\_file.close(). Make sure to .close() your file when you're done with it!

Finally, we want to know how to read from our output.txt file, we do this with the read() function:

print my\_file.read()

If you open a file and call .readline() on the file object, you'll get the first line of the file; subsequent calls to .readline() will return successive lines.

During the I/O process, data is buffered: this means that it is held in a temporary location before being written to the file.

Python doesn't flush the buffer—that is, write data to the file—until it's sure you're done writing. One way to do this is to close the file. If you write to a file without closing, the data won't make it to the target file.

file objects contain a special pair of built-in methods:  \_\_enter\_\_() and \_\_exit\_\_(). What is important is that when a file object's \_\_exit\_\_() method is invoked, it automatically closes the file. How do we invoke this method? With with and as.

The syntax looks like this:

with open("file", "mode") as variable: # Read or write to the file

with open("text.txt", "w") as textfile:

textfile.write("Success!")

Python file objects have a closed attribute which is True when the file is closed and False otherwise.

By checking file\_object.closed, we'll know whether our file is closed and can call close() on it if it's still open.

f = open("bg.txt")

f.closed # False

f.close()

f.closed # True