Chapter 6

Functions and Loops

Functions are the building blocks of almost every Python program. They're where the real action takes place!

You've already seen how to use several functions, including `print()`, `len()`, and `round()`. These are all **built-in functions** because they come built into the Python language itself. You can also create **user-defined functions** that perform specific tasks.

Functions break code into smaller chunks, and are great for tasks that a program uses repeatedly. Instead of writing the same code each time the program need to perform the task, just call the function!

But sometimes you need to repeat some code several times in a row, and this is where **loops** come in.

In this chapter, you will learn:

- How to create user-defined functions
- How to write `for` and `while` loops
- What scope is and why it is important

Let's dive in!

Leave feedback on this section »

6.1 What is a Function, Really?

In the past few chapters you used functions like `print()` and `len()` to display text and determine the length of a string. But what is a function, really?

In this section you'll take a closer look at `len()` to learn more about what a function is and how it is executed.

Functions Are Values

One of the most important properties of a function in Python is that functions are values and can be assigned to a variable.

In IDLE's interactive window, inspect the name `len` by typing the following in at the prompt:

```
>>> len
<built-in function len>
```

When you hit `Enter`, Python tells you that the name `len` is a variable whose value is a built-in function.

Just like integer values have a type called `int`, and strings have a type `str`, function values also have a type:

```
>>> type(len)
<class 'builtin_function_or_method'>
```

Like any other variable, you can assign any value you want to `len`:

```
>>> len = "I'm not the len you're looking for."
>>> len
"I'm not the len you're looking for."
```

Now `len` has a string value, and you can verify that the type is `str` with `type()`:

```
>>> type(len)
<class 'str'>
```

The variable name `len` is a keyword in Python, and even though you can change it's value, it's usually a bad idea to do so. Changing the value of `len` can make your code confusing because it's easy to mistake the new `len` for the built-in function.

Important

If you typed in the previous code examples, **you no longer have access to the built-in `len` function in IDLE**.

You can get it back with the following code:

```
>>> del len
```

The `del` keyword is used to un-assign a variable from a value. `del` stands for delete, but it doesn't delete the value. Instead, it detaches the name from the value and deletes the name.

Normally, after using `del`, trying to use the deleted variable name raises a `NameError`. In this case, however, the name `len` doesn't get deleted:

```
>>> len
<built-in function len>
```

Because `len` is a built-in function name, it gets reassigned to the original function value.

By going through each of these steps, we've seen that a function's name is separate from the function itself.

How Python Executes Functions

Now let's take a closer look at how Python executes a function.

The first thing to notice is that you can't execute a function by just

typing its name. You must **call** the function to tell Python to actually execute it.

Let's look at how this works with `len()`:

```
>>> # Typing just the name doesn't execute the function.
>>> # IDLE inspects the variable as usual.
>>> len
<built-in function len>

>>> # Use parentheses to call the function.
>>> len()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    len()
TypeError: len() takes exactly one argument (0 given)
```

In this example, Python raises a `TypeError` when `len()` is called because `len()` expects an argument.

An **argument** is a value that gets **passed** to the function as input. Some functions can be called with no arguments, and some can take as many arguments as you like. `len()` requires exactly one argument.

When a function is done executing, it **returns** a value as output. The return value usually — but not always — depends on the values of any arguments passed to the function.

The process for executing a function can be summarized in three steps:

1. The function is **called**, and any arguments are passed to the function as input.
2. The function **executes**, and some action is performed with the arguments.
3. The function **returns**, and the original function call is replaced with the return value.

Let's look at this in practice and see how Python executes the following line of code:

```
>>> num_letters = len("four")
```

First, `len()` is called with the argument `"four"`. The length of the string `"four"` is calculated, which is the number `4`. Then `len()` returns the number `4` and replaces the function call with the value.

So, after the function executes, the line of code looks like this:

```
>>> num_letters = 4
```

Then Python assigns the value `4` to `num_letters` and continues executing any remaining lines of code in the program.

Functions Can Have Side Effects

You've learned how to call a function and that they return a value when they are done executing. Sometimes, though, functions do more than just return a value.

When a function changes or affects something external to the function itself, it is said to have a **side effect**. You have already seen one function with a side effect: `print()`.

When you call `print()` with a string argument, the string is displayed in the Python shell as text. But `print()` doesn't return any text as a value.

To see what `print()` returns, you can assign the return value of `print()` to a variable:

```
>>> return_value = print("What do I return?")
What do I return?
>>> return_value
>>>
```

When you assign `print("What do I return?")` to `return_value`, the string

`"What do I return?"` is displayed. However, when you inspect the value of `return_value`, nothing is shown.

`print()` returns a special value called `None` that indicates the absence of data. `None` has a type called `NoneType`:

```
>>> type(return_value)
<class 'NoneType'>
>>> print(return_value)
None
```

When you call `print()`, the text that gets displayed is not the return value. It is a side effect of `print()`.

Now that you know that functions are values, just like strings and numbers, and have learned how functions are called and executed, let's take a look at how you can create your own user-defined functions.

Leave feedback on this section »

6.2 Write Your Own Functions

As you write longer and more complex programs, you may find that you need to use the same few lines of code repeatedly. Or maybe you need to calculate the same formula with different values several times in your code.

You might be tempted to copy and paste similar code to other parts of your program and modify it as needed, but this is usually a bad idea!

Repetitive code can be a nightmare to maintain. If you find a mistake in some code that's been copied and pasted all over the place, you'll end up having to apply the fix everywhere the code was copied. That's a lot of work, and you might miss a spot!

In this section, you'll learn how to define your own functions so that you can avoid repeating yourself when you need to reuse code. Let's

go!

The Anatomy of a Function

Every function has two parts:

1. The **function signature** defines the name of the function and any inputs it expects.
2. The **function body** contains the code that runs every time the function is used.

Let's start by writing a function that takes two numbers as input and returns their product. Here's what this function might look like, with the signature, body, and return statement identified with comments:

```
def multiply(x, y):  # Function signature
    # Function body
    product = x * y
    return product
```

It might seem odd to make a function for something as simple as the `*` operator. In fact, `multiply` is not a function you would probably write in a real-world scenario. But it makes a great first example for understanding how functions are created!

Let's break the function down to see what's going on.

The Function Signature

The first line of code in a function is called the **function signature**. It always starts with the `def` keyword, which is short for "define."

Let's look more closely at the signature of the `multiply` function:

```
def multiply(x, y):
```

The function signature has four parts:

1. The `def` keyword

2. The function name, `multiply`
3. The parameter list, `(x, y)`
4. A colon (`:`) at the end of the line

When Python reads a line beginning with the `def` keyword, it creates a new function. The function is assigned to a variable with the same name as the function name.

Note

Since function names become variables, they must follow the same rules for variable names that you learned in Chapter 3.

So, a function name can only contain numbers, letters, and underscores, and must not begin with a number.

The parameter list is a list of parameter names surrounded by opening and closing parentheses. It defines the function's expected inputs. `(x, y)` is the parameter list for the `multiply` function. It creates two parameters, `x` and `y`.

A **parameter** is sort of like a variable, except that it has no value. It is a placeholder for actual values that are provided whenever the function is called with one or more arguments.

Code in the function body can use parameters as if they are variables with real values. For example, the function body may contain a line of code with the expression `x * y`.

Since `x` and `y` have no value, `x * y` has no value. Python saves the expression as a template and fills in the missing values when the function is executed.

A function can have any number of parameters, including no parameters at all!

The Function Body

The **function body** is the code that gets run whenever the function is used in your program. Here's the function body for the `multiply` function:

```
def multiply(x, y):
    # Function body
    product = x * y
    return product
```

`multiply` is a pretty simple function. It's body has only two lines of code!

The first line creates a variable called `product` and assigns to it the value `x * y`. Since `x` and `y` have no values yet, this line is really a template for the value `product` is assigned when the function is executed.

The second line of code is called a **return statement**. It starts with the `return` keyword and is followed by the variable `product`. When Python reaches the return statement, it stops running the function and returns the value of `product`.

Notice that both lines of code in the function body are indented. This is vitally important! Every line that is indented below the function signature is understood to be part of the function body.

For instance, the `print()` function in the following example is not a part of the function body because it is not indented:

```
def multiply(x, y):
    product = x * y
    return product

print("Where am I?")  # Not in the function body.
```

If `print()` is indented, then it becomes a part of the function body even if there is a blank line between `print()` and the previous line:

```
def multiply(x, y):
    product = x * y
    return product

    print("Where am I?")  # In the function body.
```

There is one rule that you must follow when indenting code in a function's body. Every line must be indented by the same number of spaces.

Try saving the following code to a file called `multiply.py` and running it from IDLE:

```
def multiply(x, y):
    product = x * y
     return product  # Indented with one extra space.
```

IDLE won't run the code! A dialog box appears with the error "unexpected indent." Python wasn't expecting the return statement to be indented differently than the line before it.

Another error occurs when a line of code is indented less than the line above it, but the indentation doesn't match any previous lines. Modify the `multiply.py` file to look like this:

```
def multiply(x, y):
     product = x * y
    return product  # Indented less than previous line.
```

Now save and run the file. IDLE stops it with the error "unindent does not match any outer indentation level." The return statement isn't indented with the same number of spaces as any other line in the function body.

Note

Although Python has no rules for the number of spaces used to indent code in a function body, PEP 8 recommends indenting with four spaces.

We follow this convention throughout this book.

Once Python executes a `return` statement, the function stops running and returns the value. If any code appears below the `return` statement that is indented so as to be part of the function body, it will never run.

For instance, the `print()` function will never be executed in the following function:

```
def multiply(x, y):
    product = x * y
    return product
    print("You can't see me!")
```

If you call this version of `multiply()`, you will never see the string `"You can't see me!"` displayed.

Calling a User-Defined Function

You call a user-defined function just like any other function. Type the function name followed by a list of arguments in between parentheses.

For instance, to call `multiply()` with the argument `2` and `4`, just type:

```
multiply(2, 4)
```

Unlike built-in functions, user-defined functions are not available until they have been defined with the `def` keyword. You must define the function before you call it.

Try saving and running the following script:

```
num = multiply(2, 4)
print(num)

def multiply(x, y):
    product = x * y
    return product
```

When Python reads the line `num = multiply(2, 4)`, it doesn't recognize the name `multiply` and raises a `NameError`:

```
Traceback (most recent call last):
  File "C:Usersdaveamultiply.py", line 1, in <module>
    num = multiply(2, 4)
NameError: name 'multiply' is not defined
```

To fix the error, move the function definition to the top of the file:

```
def multiply(x, y):
    product = x * y
    return product

num = multiply(2, 4)
print(num)
```

Now when you save and run the script, the value `8` is displayed in the interactive window.

Functions With No Return Statement

All functions in Python return a value, even if that value is `None`. However, not all functions need a `return` statement.

For example, the following function is perfectly valid:

```
def greet(name):
    print(f"Hello, {name}!")
```

`greet()` has no `return` statement, but works just fine:

```
>>> greet("Dave")
Hello, Dave!
```

Even though `greet()` has no `return` statement, it still returns a value:

```
>>> return_value = greet("Dave")
Hello, Dave!
>>> print(return_value)
None
```

Notice also that the string `"Hello, Dave!"` is printed even when the result of `greet("Dave")` is assigned to a variable. That's because the call to `print()` inside of the `greet()` function body produces the side effect of always printing to the console.

If you weren't expecting to see `"Hello, Dave!"` printed, then you just experienced one of the issues with side effects. They aren't always expected!

When you create your own functions, you should always document what they do. That way other developers can read the documentation and know how to use the function and what to expect when it is called.

Documenting Your Functions

To get help with a function in IDLE's interactive window, you can use the `help()` function:

```
>>> help(len)
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

When you pass a variable name or function name to `help()`, it displays some useful information about it. In this case, `help()` tells you that `len` is a built-in function that returns the number of items in a container.

Note

A **container** is a special name for an object that contains other objects. A string is a container because it contains characters.

You will learn about other container types in Chapter 9.

Let's see what happens when you call `help()` on the `multiply()` function:

```
>>> help(multiply)
Help on function multiply in module __main__:

multiply(x, y)
```

`help()` displays the function signature, but there isn't any information about what the function does. To better document `multiply()`, we need to provide a docstring.

A **docstring** is a triple-quoted string literal placed at the top of the function body. Docstrings are used to document what a function does and what kinds of parameters it expects.

Here's what `multiply()` looks like with a docstring added to it:

```
def multiply(x, y):
    """Return the product of two numbers x and y."""
    product = x * y
    return product
```

Update the `multiply.py` script with the docstring, then save and run the script. Now you can use `help()` in the interactive window to see the docstring:

```
>>> help(multiply)
Help on function multiply in module __main__:

multiply(x, y)
    Return the product of two numbers x and y.
```

PEP 8 doesn't say much about docstrings, except that every function should have one.

There are a number of standardized docstring formats, but we won't get into them here. Some general guidelines for writing docstrings can be found in PEP 257.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Write a function called `cube()` with one number parameter and returns the value of that number raised to the third power. Test the function by displaying the result of calling your `cube()` function on a few different numbers.
2. Write a function called `greet()` that takes one string parameter called `name` and displays the text `"Hello <name>!"`, where `<name>` is replaced with the value of the `name` parameter.

Leave feedback on this section »

6.3 Challenge: Convert Temperatures

Write a script called `temperature.py` that defines two functions:

1. `convert_cel_to_far()` which takes one `float` parameter representing degrees Celsius and returns a float representing the same temperature in degrees Fahrenheit using the following formula:

   ```
   F = C * 9/5 + 32
   ```

2. `convert_far_to_cel()` which take one `float` parameter representing degrees Fahrenheit and returns a float representing the same temperature in degrees Celsius using the following formula:

   ```
   C = (F - 32) * 5/9
   ```

The script should first prompt the user to enter a temperature in degrees Fahrenheit and then display the temperature converted to Celsius.

Then prompt the user to enter a temperature in degrees Celsius and display the temperature converted to Fahrenheit.

All converted temperatures should be rounded to 2 decimal places.

Here's a sample run of the program:

```
Enter a temperature in degrees F: 72
72 degrees F = 22.22 degrees C

Enter a temperature in degrees C: 37
37 degrees C = 98.60 degrees F
```

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources.

Leave feedback on this section »

6.4 Run in Circles

One of the great things about computers is that you can make them do the same thing over and over again, and they rarely complain or get tired.

A **loop** is a block of code that gets repeated over and over again either a specified number of times or until some condition is met. There are two kinds of loops in Python: **`while` loops** and **`for` loops**. In this section, you'll learn how to use both.

Let's start by looking at how `while` loops work.

The `while` Loop

`while` loops repeat a section of code while some condition is true. There are two parts to every `while` loop:

1. The **`while` statement** starts with the `while` keyword, followed by a **test condition**, and ends with a colon (:).
2. The **loop body** contains the code that gets repeated at each step of the loop. Each line is indented four spaces.

When a `while` loop is executed, Python evaluates the test condition and determines if it is true or false. If the test condition is true, then the code in the loop body is executed. Otherwise, the code in the body is skipped and the rest of the program is executed.

If the test condition is true and the body of the loop is executed, then once Python reaches the end of the body, it returns to the `while` statement and re-evaluates the test condition. If the test condition is still true, the body is executed again. If it is false, the body is skipped.

This process repeats over and over until the test condition fails, causing Python to loop over the code in the body of the `while` loop.

Let's look at an example. Type the following code into the interactive window:

```
>>> n = 1
>>> while n < 5:
...     print(n)
...     n = n + 1
...
1
2
3
4
```

First, the integer `1` is assigned to the variable `n`. Then a `while` loop is created with the test condition `n < 5`, which checks whether or not the value of `n` is less than `5`.

If `n` is less than `5`, the body of the loop is executed. There are two lines of code in the loop body. In the first line, the value of `n` is printed on the screen, and then `n` is **incremented** by `1` in the second line.

The loop execution takes place in five steps, described in the following table:

Step #	Value of `n`	Test Condition	What Happens
1	`1`	`1 < 5` (true)	`1` printed; `n` incremented to `2`
2	`2`	`2 < 5` (true)	`2` printed; `n` incremented to `3`
3	`3`	`3 < 5` (true)	`3` printed; `n` incremented to `4`
4	`4`	`4 < 5` (true)	`4` printed; `n` incremented to `5`
5	`5`	`5 < 5` (false)	Nothing printed; loop ends.

If you aren't careful, you can create an **infinite loop**. This happens when the test condition is always true. An infinite loop never terminates. The loop body keeps repeating forever.

Here's an example of an infinite loop:

```
>>> n = 1
>>> while n < 5:
...     print(n)
...
```

The only difference between this `while` loop and the previous one is that `n` is never incremented in the loop body. At each step of the loop, `n` is equal to `1`. That means the test condition `n < 5` is always true, and the number `1` is printed over and over again forever.

Note

Infinite loops aren't inherently bad. Sometimes they are exactly the kind of loop you need.

For example, code that interacts with hardware may use an infinite loop to constantly check whether or not a button or switch has been activated.

If you run a program that enters an infinite loop, you can force Python to quit by pressing `Ctrl+C`. Python stops running the program and raises a `KeyboardInterrupt` error:

```
Traceback (most recent call last):
  File "<pyshell#8>", line 2, in <module>
    print(n)
KeyboardInterrupt
```

Let's look at an example of a `while` loop in practice. One use of a `while` loop is to check whether or not user input meets some condition and, if not, repeatedly ask the user for new input until valid input is received.

For instance, the following program continuously asks a user for a positive number until a positive number is entered:

```
num = float(input("Enter a positive number: "))

while num <= 0:
    print("That's not a positive number!")
    num = float(input("Enter a positive number: "))
```

First, the user is prompted to enter a positive number. The test condition `num <= 0` determines whether or not `num` is less than or equal to `0`.

If `num` is positive, then the test condition fails. The body of the loop is skipped and the program ends.

Otherwise, if `num` is `0` or negative, the body of the loop executes. The

user is notified that their input was incorrect, and they are prompted again to enter a positive number.

`while` loops are perfect for repeating a section of code while some condition is met. They aren't well-suited, however, for repeating a section of code a specific number of times.

The `for` Loop

A `for` loop executes a section of code once for each item in a collection of items. The number of times that the code is executed is determined by the number of items in the collection.

Like its `while` counterpart, the `for` loop has two main parts:

1. The **`for` statement** begins with the `for` keyword, followed by a **membership expression**, and ends in a colon (:).
2. The **loop body** contains the code to be executed at each step of the loop, and is indented four spaces.

Let's look at an example. The following `for` loop prints each letter of the string `"Python"` one at a time:

```
for letter in "Python":
    print(letter)
```

In this example, the `for` statement is `for letter in "Python"`. The membership expression is `letter in "Python"`.

At each step of the loop, the variable `letter` is assigned the next letter in the string `"Python"`, and then the value of `letter` is printed.

The loops runs once for each character in the string `"Python"`, so the loop body executes six times. The following table summarizes the execution of this `for` loop:

Step #	Value of `letter`	What Happens
1	`"P"`	`P` is printed

Step #	Value of `letter`	What Happens
2	`"y"`	`y` is printed
3	`"t"`	`t` is printed
4	`"h"`	`h` is printed
5	`"o"`	`o` is printed
6	`"n"`	`n` is printed

To see why `for` loops are better for looping over collections of items, let's re-write the `for` loop in previous example as a `while` loop.

To do so, we can use a variable to store the index of the next character in the string. At each step of the loop, we'll print out the character at the current index and then increment the index.

The loop will stop once the value of the index variable is equal to the length of the string. Remember, indices start at `0`, so the last index of the string `"Python"` is `5`.

Here's how you might write that code:

```
word = "Python"
index = 0

while index < len(word):
    print(word[index])
    index = index + 1
```

That's significantly more complex than the `for` loop version!

Not only is the `for` loop less complex, the code itself looks more natural. It more closely resembles how you might describe the loop in English.

Note

You may sometimes hear people describe some code as being particularly "Pythonic." The term **Pythonic** is generally used to describe code that is clear, concise, and uses Python's built-in features to its advantage.

In these terms, using a `for` loop to loop over a collection of items is more Pythonic than using a `while` loop.

Sometimes it's useful to loop over a range of numbers. Python has a handy built-in function `range()` that produces just that — a range of numbers!

For example, `range(3)` returns the range of integers starting with `0` and up to, but not including, `3`. That is, `range(3)` is the range of numbers `0`, `1`, and `2`.

You can use `range(n)`, where `n` is any positive number, to execute a loop exactly `n` times. For instance, the following `for` loop prints the string `"Python"` three times:

```
for n in range(3):
    print("Python")
```

You can also give a range a starting point. For example, `range(1, 5)` is the range of numbers `1`, `2`, `3`, and `4`. The first argument is the starting number, and the second argument is the endpoint, which is not included in the range.

Using the two-argument version of `range()`, the following `for` loop prints the square of every number starting with `10` and up to, but not including, `20`:

```
for n in range(10, 20):
    print(n * n)
```

Let's look at a practical example. The following program asks the user to input an amount and then displays how to split that amount be-

tween 2, 3, 4, and 5 people:

```
amount = float(input("Enter an amount: "))

for num_people in range(2, 6):
    print(f"{num_people} people: ${amount / num_people:,.2f} each")
```

The `for` loop loops over the number `2`, `3`, `4`, and `5`, and prints the number of people and the amount each person should pay. The formatting specifier `,.2f` is used to format the amount as fixed-point number rounded to two decimal places and commas every three digits.

Running the program with the input `10` produces the following output:

```
Enter an amount: 10
2 people: $5.00 each
3 people: $3.33 each
4 people: $2.50 each
5 people: $2.00 each
```

`for` loops are generally used more often than `while` loops in Python. Most of the time, a `for` loop is more concise and easier to read than an equivalent `while` loop.

Nested Loops

As long as you indent the code correctly, you can even put loops inside of other loops.

Type the following into IDLE's interactive window:

```
for n in range(1, 4):
    for j in range(4, 7):
        print(f"n = {n} and j = {j}")
```

When Python enters the body of the first `for` loop, the variable `n` is assigned the value `1`. Then the body of the second `for` loop is executed and `j` is assigned the value `4`. The first thing printed is `n = 1 and j = 4`.

After executing the `print()` function, Python returns to the *inner* `for` loop, assigns to `j` the value of `5`, and then prints `n = 1 and j = 5`. Python doesn't return the outer `for` loop because the inner `for` loop, which is inside the body of the outer `for` loop, isn't done executing.

Next, `j` is assigned the value `6` and Python prints `n = 1 and j = 6`. At this point, the inner `for` loop is done executing, so control returns to the outer `for` loop.

The variable `n` gets assigned the value `2`, and the inner `for` loop executes a second time. That is, `j` is assigned the value `4` and `n = 2 and j = 4` is printed to the console.

The two loops continue to execute in this fashion, and the final output looks like this:

```
n = 1 and j = 4
n = 1 and j = 5
n = 1 and j = 6
n = 2 and j = 4
n = 2 and j = 5
n = 2 and j = 6
n = 3 and j = 4
n = 3 and j = 5
n = 3 and j = 6
```

A loop inside of another loop is called a **nested loop**, and they come up more often than you might expect. You can nest `while` loops inside of `for` loops, and vice versa, and even nest loops more than two levels deep!

Important

Nesting loops inherently increases the complexity of your code, as you can see by the dramatic increase in the number of steps run in the previous example compared to examples with a single `for` loop.

Using nested loops is sometimes the only way to get something done, but too many nested loops can have a negative effect on a program's performance.

Loops are a powerful tool. They tap into one of the greatest advantages computers provide as tools for computation: the ability to repeat the same task a vast number of times without tiring and without complaining.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Write a `for` loop that prints out the integers 2 through 10, each on a new line, by using the `range()` function.
2. Use a `while` loop that prints out the integers 2 through 10 (Hint: You'll need to create a new integer first.)
3. Write a function called `doubles()` that takes one number as its input and doubles that number. Then use the `doubles()` function in a loop to double the number `2` three times, displaying each result on a separate line. Here is some sample output:

```
4
8
16
```

Leave feedback on this section »

6.5 Challenge: Track Your Investments

In this challenge, you will write a program called `invest.py` that tracks the growing amount of an investment over time.

An initial deposit, called the principal amount, is made. Each year, the amount increases by a fixed percentage, called the annual rate of return.

For example, a principal amount of $100 with an annual rate of return of 5% increases the first year by $5. The second year, the increase is 5% of the new amount $105, which is $5.25.

Write a function called `invest` with three parameters: the principal amount, the annual rate of return, and the number of years to calculate. The function signature might look something like this:

```
def invest(amount, rate, years):
```

The function then prints out the amount of the investment, rounded to 2 decimal places, at the end of each year for the specified number of years.

For example, calling `invest(100, .05, 4)` should print the following:

```
year 1: $105.00
year 2: $110.25
year 3: $115.76
year 4: $121.55
```

To finish the program, prompt the user to enter an initial amount, an annual percentage rate, and a number of years. Then call `invest()` to display the calculations for the values entered by the user.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources.

Leave feedback on this section »

6.6 Understand Scope in Python

Any discussion of functions and loops in Python would be incomplete without some mention of the issue of **scope**.

Scope can be one of the more difficult concepts to understand in programming, so in this section you will get a gentle introduction to it.

By the end of this section, you'll know what a **scope** is and why it is important. You will also learn the LEGB rule for **scope resolution**.

What Is a Scope?

When you assign a value to a variable, you are giving that value a name. Names are unique. For example, you can't assign the same name to two different numbers.

```
>>> x = 2
>>> x
2

>>> x = 3
>>> x
3
```

When you assign `3` to `x`, you can no longer recall the value `2` with the name `x`.

This behavior makes sense. After all, if the variable `x` has the values `2` and `3` simultaneously, how do you evaluate `x + 2`? Should it be `4` or `5`?

As it turns out, there *is* a way to assign the same name to two different values. Try running the following script:

```
x = "Hello World"

def func():
    x = 2
```

```
    print(f"Inside 'func', x has the value {x}")

func()
print(f"Outside 'func', x has the value {x}")
```

In this example, the variable `x` is assigned two different values. `x` is assigned `"Hello, World"` at the beginning, and is assigned `2` inside of `func()`.

The output of the script, which you might find surprising, looks like this:

```
Inside 'func', x has the value 2
Outside 'func', x has the value Hello World
```

How does `x` still have the value `"Hello World"` after calling `func()`, which changes the value of `x` to `2`?

The answer is that the function `func()` has a different **scope** than the code that exists outside of the function. That is, you can name an object inside `func()` the same name as something outside `func()` and Python can keep the two separated.

The function body has what is known as a **local scope**, with its own set of names available to it. Code outside of the function body is in the **global scope**.

You can think of a scope as a set of names mapped to objects. When you use a particular name in your code, such as a variable or a function name, Python checks the current scope to determine whether or not that name exists.

Scope Resolution

Scopes have a hierarchy. For example, consider the following:

```
x = 5
```

```
def outer_func():
    y = 3

    def inner_func():
        z = x + y
        return z

    return inner_func()
```

Note

The `inner_func()` function is called an **inner function** because it is defined inside of another function. Just like you can nest loops, you can also define functions within other functions!

You can read more about inner functions in Real Python's article Inner Functions—What Are They Good For?.

The variable `z` is in the local scope of `inner_func()`. When Python executes the line `z = x + y`, it looks for the variables `x` and `y` in the local scope. However, neither of them exist there, so it moves up to the scope of the `outer_func()` function.

The scope for `outer_func()` is an **enclosing** scope of `inner_func()`. It is not quite the global scope, and is not the local scope for `inner_func()`. It lies in between those two.

The variable `y` is defined in the scope for `outer_func()` and is assigned the value `3`. However, `x` does not exist in this scope, so Python moves up once again to the global scope. There it finds the name `x`, which has the value `5`. Now that the names `x` and `y` are resolved, Python can execute the line `z = x + y`, which assigns to `z` the value of `8`.

The LEGB Rule

A useful way to remember how Python resolves scope is with the **LEGB** rule. This rule is an acronym for **L**ocal, **E**nclosing, **G**lobal, **B**uilt-in.

Python resolves scope in the order in which each scope appears in the list LEGB. Here is a quick overview to help you remember how all of this works:

Local (L): The local, or current, scope. This could be the body of a function or the top-level scope of a script. It always represents the scope that the Python interpreter is currently working in.

Enclosing (E): The enclosing scope. This is the scope one level up from the local scope. If the local scope is an inner function, the enclosing scope is the scope of the outer function. If the scope is a top-level function, the enclosing scope is the same as the global scope.

Global (G): The global scope, which is the top-most scope in the script. This contains all of the names defined in the script that are not contained in a function body.

Built-in (B): The built-in scope contains all of the names, such as keywords, that are built-in to Python. Functions such as `round()` and `abs()` are in the built-in scope. Anything that you can use without first defining yourself is contained in the built-in scope.

Break the Rules

Consider the following script. What do you think the output is?

```
total = 0

def add_to_total(n):
    total = total + n

add_to_total(5)
print(total)
```

You would think that script outputs the value `5`, right? Try running it to see what happens.

Something unexpected occurs. You get an error!

```
Traceback (most recent call last):
  File "C:/Users/davea/stuff/python/scope.py", line 6, in <module>
    add_to_total(5)
  File "C:/Users/davea/stuff/python/scope.py", line 4, in add_to_total
    total = total + n
UnboundLocalError: local variable 'total' referenced before assignment
```

Wait a minute! According to the LEGB rule, Python should have recognized that the name `total` doesn't exist in the `add_to_total()` function's local scope and moved up to the global scope to resolve the name, right?

The problem here is that the script attempts to make an assignment to the variable `total`, which creates a new name in the local scope. Then, when Python executes the right-hand side of the assignment it finds the name `total` in the local scope with nothing assigned to it yet.

These kinds of errors are tricky and are one of the reasons it is best to use unique variable and function names no matter what scope you are in.

You can get around this issue with the `global` keyword. Try running the following altered script:

```
total = 0

def add_to_total(n):
    global total
    total = total + n

add_to_total(5)
print(total)
```

This time, you get the expected output `5`. Why's that?

The line `global total` tells Python to look in the global scope for the name `total`. That way, the line `total = total + n` does not create a new local variable.

Although this "fixes" the script, the use of the `global` keyword is considered bad form in general.

If you find yourself using `global` to fix problems like the one above, stop and think if there is a better way to write your code. Often, you'll find that there is!

Leave feedback on this section »

6.7 Summary and Additional Resources

In this chapter, you learned about two of the most essential concepts in programming: functions and loops.

First, you learned how to define your own custom functions. You saw that functions are made up of two parts:

1. The **function signature**, which starts with the `def` keyword and includes the name of the function and the function's parameters
2. The **function body**, which contains the code that runs whenever the function is called.

Functions help avoid repeating similar code throughout a program by creating re-usable components. This helps make code easier to read and maintain.

Then you learned about Python's two kinds of loops:

1. **`while` loops** repeat some code while some condition remains true
2. **`for` loops** repeat some code for each element in a set of objects

Finally, you learned what a **scope** is and how Python resolves scope using the LEGB rule.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/python-basics-6

Additional Resources

To learn more about functions and loops, check out the following resources:

- Python “while” Loops (Indefinite Iteration)
- Python “for” Loops (Definite Iteration)
- Recommended resources on realpython.com

Leave feedback on this section »