Important

When you write code in a script, you do not need to include the `>>>` prompt that you see in IDLE's interactive window. Keep this in mind if you copy and paste code from examples that show the REPL prompt.

Remember, though, that it's not recommended that you copy and paste examples from the book. Typing each example in yourself really pays off!

Before you can run your script, you must save it. From the menu at the top of the window, select File > Save and save the script as `hello_world.py`. The `.py` file extension is the conventional extension used to indicate that a file contains Python code.

In fact, if you save your script with any extension other than `.py`, the code highlighting will disappear and all the text in the file will be displayed in black. IDLE will only highlight Python code when it is stored in a `.py` file.

Once the script is saved, all you have to do to run the program is select Run > Run Module from the script window and you'll see `Hello, world` appear in the interactive window:

```
Hello, world
```

Note

You can also press F5 to run a script from the script window.

Every time you run a script you will see something like the following output in the interactive window:

```
>>> ================== RESTART ==================
```

This is IDLE's way of separating output from distinct runs of a script. Otherwise, if you run one script after another, it may not be clear what

output belongs to which script.

To open an existing script in IDLE, select File » Open... from the menu in either the script window or the interactive window. Then browse for and select the script file you want to open. IDLE opens scripts in a new script window, so you can have several scripts open at a time.

Note

Double-clicking on a `.py` file from a file manager, such as Windows Explorer, does execute the script in a new window. However, the window is closed immediately when the script is done running—often before you can even see what happened.

To open the file in IDLE so that you can run it and see the output, you can right-click on the file icon (Ctrl + Click on macOS) and choose to Edit with IDLE.

Leave feedback on this section »

3.2 Mess Things Up

Everybody makes mistakes—especially while programming! In case you haven't made any mistakes yet, let's get a head start on that and mess something up on purpose to see what happens.

Mistakes made in a program are called **errors**, and there are two main types of errors you'll experience:

1. Syntax errors
2. Run-time errors

In this section you'll see some examples of code errors and learn how to use the output Python displays when an error occurs to understand what error occurred and which piece of code caused it.

Syntax Errors

In loose terms, a **syntax error** occurs when you write some code that isn't allowed in the Python language. You can create a syntax error by changing the contents of the `hello_world.py` script from the last section to the following:

```
print("Hello, world)
```

In this example, the double quotation mark at the end of `"Hello, world"` has been removed. Python won't be able to tell where the string of text ends. Save the altered script and then try to run it. What happens?

The code won't run! IDLE displays an alert box with the following message:

```
EOL while scanning string literal.
```

EOL stands for **E**nd **O**f **L**ine, so this message tells you that Python read all the way to the end of the line without finding the end of something called a string literal.

A **string literal** is text contained in-between two double quotation marks. The text `"Hello, world"` is an example of a string literal.

Note

For brevity, string literals are often referred to as **strings**, although the term "string" technically has a more general meaning in Python. You will learn more about strings in Chapter 4.

Back in the script window, notice that the line containing with `"Hello, world` is highlighted in red. This handy features helps you quickly find which line of code caused the syntax error.

Run-time Errors

IDLE catches syntax errors before a program starts running, but some errors can't be caught until a program is executed. These errors are

known as **run-time errors** because they only occur at the time that a program is run.

To generate a run-time error, change the code in `hello_world.py` to the following:

```
print(Hello, world)
```

Now both quotation marks from the phrase `"Hello, world"` have been removed. Did you notice how the text color changes to black when you removed the quotation marks? IDLE no longer recognizes `Hello, world` as a string.

What do you think happens when you run the script? Try it out and see!

Some red text is displayed in the interactive window:

```
Traceback (most recent call last):
  File "/home/hello_world.py", line 1, in <module>
    print(Hello, world)
NameError: name 'Hello' is not defined
```

What happened? While trying to execute the program Python **raised** an error. Whenever an error occurs, Python stops executing the program and displays the error in IDLE's interactive window.

The text that gets displayed for an error is called a **traceback**. Tracebacks give you some useful information about the error. The traceback above tells us all of the following:

- The error happened on line `1` of the `hello_world.py`.
- The line that generated the error was: `print(Hello, world)`.
- A `NameError` occurred.
- The specific error was `name 'Hello' is not defined`

The quotation marks around `Hello, world` are missing, so Python doesn't understand that it is a string of text. Instead, Python thinks

that `Hello` and `world` are the names of something else in the code. Since names `Hello` and `world` haven't been defined anywhere, the program crashes.

In the next section, you'll see how to define names for values in your code. Before you move on though, you can get some practice with syntax errors and run-time errors by working on the review exercises.

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 script that IDLE won't let you run because it has a syntax error.
2. Write a script that only crashes your program once it is already running because it has a run-time error.

Leave feedback on this section »

3.3 Create a Variable

In Python, **variables** are names that can be assigned a value and used to reference that value throughout your code. Variables are fundamental to programming for two reasons:

1. **Variables keep values accessible:** For example, the result of some time-consuming operation can be assigned to a variable so that the operation does not need to be performed each time you need to use the result.
2. **Variables give values context:** The number `28` could mean lots of different things, such as the number of students in a class, or the number of times a user has accessed a website, and so on. Naming the value `28` something like `num_students` makes the meaning of the value clear.

In this section, you'll learn how to use variables in your code, as well as some of the conventions Python programmers follow when choosing names for variables.

The Assignment Operator

Values are assigned to a variable using a special symbol `=` called the **assignment operator**. An **operator** is a symbol, like `=` or `+`, that performs some operation on one or more values.

For example, the `+` operator takes two numbers, one to the left of the operator and one to the right, and adds them together. Likewise, the `=` operator takes a value to the right of the operator and assigns it to the name on the left of the operator.

To see the assignment operator in action, let's modify the "Hello, world" program you saw in the last section. This time, we'll use a variable to store some text before printing it to the screen:

```
>>> phrase = "Hello, world"
>>> print(phrase)
Hello, world
```

In the first line, a variable named `phrase` is created and assigned the value `"Hello, world"` using the `=` operator. The string `"Hello, world"` that was originally used inside of the parentheses in the `print()` function is replaced with the variable `phrase`.

The output `Hello, world` is displayed when you execute `print(phrase)` because Python looks up the name `phrase` and finds it has been assigned the value `"Hello, world"`.

If you hadn't executed `phrase = "Hello, world"` before executing `print(phrase)`, you would have seen a `NameError` like you did when trying to execute `print(Hello, world)` in the previous section.

Note

Although = looks like the equals sign from mathematics, it has a different meaning in Python. Distinguishing the = operator from the equals sign is important, and can be a source of frustration for beginner programmers.

Just remember, whenever you see the = operator, whatever is to the right of it is being assigned to a variable on the left.

Variable names are **case-sensitive**, so a variable named `phrase` is distinct from a variable named `Phrase` (note the capital `P`). For instance, the following code produces a `NameError`:

```
>>> phrase = "Hello, world"
>>> print(Phrase)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Phrase' is not defined
```

When you run into trouble with the code examples in this book, be sure to double-check that every character in your code—including spaces—exactly matches the examples. Computers can't use common sense to interpret what you meant to say, so being *almost* correct won't get a computer to do the right thing!

Rules for Valid Variable Names

Variable names can be as long or as short as you like, but there are a couple of rules that you must follow. Variable names can only contain uppercase and lowercase letters (A–Z, a–z), digits (0–9), and underscores (_). However, variable names cannot begin with a digit.

For example, `phrase`, `string1`, `_a1p4a`, and `list_of_names` are all valid variable names, but `9lives` is not.

Note

Python variable names can contain many different valid Unicode characters. **Unicode** is a standard for digitally representing text used in most of the world's writing systems.

That means variable names can contain letters from non-English alphabets, such as decorated letters like é and ü, and even Chinese, Japanese, and Arabic symbols.

However, not every system can display decorated characters, so it is a good idea to avoid them if your code is going to be shared with people in many different regions.

You can learn more about Unicode on Wikipedia. Python's support for Unicode is covered in the official Python documentation.

Just because a variable name is valid doesn't necessarily mean that it is a good name. Choosing a good name for a variable can be surprisingly difficult. However, there are some guidelines that you can follow to help you choose better names.

Descriptive Names Are Better Than Short Names

Descriptive variable names are essential, especially for complex programs. Often, descriptive names require using multiple words. Don't be afraid to use long variable names.

In the following example, the value `3600` is assigned to the variable `s`:

```
s = 3600
```

The name `s` is totally ambiguous. Using a full word makes it a lot easier to understand what the code means:

```
seconds = 3600
```

`seconds` is a better name than `s` because it provides more context. But

it still doesn't convey the full meaning of the code. Is `3600` the number of seconds it takes for some process to finish, or the length of a movie? There's no way to tell.

The following name leaves no doubt about what the code means:

```
seconds_per_hour = 3600
```

When you read the above code, there is no question that `3600` is the number of seconds in one hour. Although `seconds_per_hour` takes longer to type than both the single letter `s` and the word `seconds`, the pay-off in clarity is massive.

Although naming variables descriptively means using longer variable names, you should avoid names that are excessively long. What "excessively long" really means is subjective, but a good rule of thumb is to keep variable names to fewer than three or four words.

Python Variable Naming Conventions

In many programming languages, it is common to write variable names in **camelCase** like `numStudents` and `listOfNames`. The first letter of every word, except the first, is capitalized, and all other letters are lowercase. The juxtaposition of lower-case and upper-case letters look like humps on a camel.

In Python, however, it is more common to write variable names in **snake case** like `num_students` and `list_of_names`. Every letter is lowercase, and each word is separated by an underscore.

While there is no hard-and-fast rule mandating that you write your variable names in snake case, the practice is codified in a document called PEP 8, which is widely regarded as the official style guide for writing Python.

Following the standards outlined in PEP 8 ensures that your Python code is readable by a large number of Python programmers. This makes sharing and collaborating on code easier for everyone involved.

Note

All of the code examples in this course follow PEP 8 guidelines, so you will get a lot of exposure to what Python code that follows standard formatting guidelines looks like.

In this section you learned how to create a variable, rules for valid variable names, and some guidelines for choosing good variable names. Next, you will learn how to inspect a variable's value in IDLE's interactive window.

Review Exercises

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

1. Using the interactive window, display some text on the screen by using the `print()` function.
2. Using the interactive window, display a string of text by saving the string to a variable, then reference the string in a `print()` function using the variable name.
3. Do each of the first two exercises again by first saving your code in a script and running it.

Leave feedback on this section »

3.4 Inspect Values in the Interactive Window

You have already seen how to use `print()` to display a string that has been assigned to a variable. There is another way to display the value of a variable when you are working in the Python shell.

Type the following into IDLE's interactive window:

```
>>> phrase = "Hello, world"
>>> phrase
```

When you press `Enter` after typing `phrase` a second time, the following output is displayed:

```
'Hello, world'
```

Python prints the string `"Hello, world"`, and you didn't have to type `print(phrase)`!

Now type the following:

```
>>> print(phrase)
```

This time, when you hit `Enter` you see:

```
Hello, world
```

Do you see the difference between this output and the output of simply typing `phrase`? It doesn't have any single quotes surrounding it. What's going on here?

When you type `phrase` and press `Enter`, you are telling Python to **inspect** the variable `phrase`. The output displayed is a useful representation of the value assigned to the variable.

In this case, `phrase` is assigned the string `"Hello, world"`, so the output is surrounded with single quotes to indicate that `phrase` is a string.

On the other hand, when you `print()` a variable, Python displays a more human-readable representation of the variable's value. For strings, both ways of being displayed are human-readable, but this is not the case for every type of value.

Sometimes, both printing and inspecting a variable produces the same output:

```
>>> x = 2
>>> x
2
>>> print(x)
2
```

Here, `x` is assigned to the number `2`. Both the output of `print(x)` and inspecting `x` is not surrounded with quotes, because `2` is a number and not a string.

Inspecting a variable, instead of printing it, is useful for a couple of reasons. You can use it to display the value of a variable without typing `print()`. More importantly, though, inspecting a variable usually gives you more useful information than `print()` does.

Suppose you have two variables: `x = 2` and `y = "2"`. In this case, `print(x)` and `print(y)` both display the same thing. However, inspecting `x` and `y` shows the difference between the each variable's value:

```
>>> x = 2
>>> y = "2"
>>> print(x)
2
>>> print(y)
2
>>> x
2
>>> y
'2'
```

The key takeaway here is that `print()` displays a readable representation of a variable's value, while inspection provides additional information about the type of the value.

You can inspect more than just variables in the Python shell. Check out what happens when you type `print` and hit `Enter`:

```
>>> print
<built-in function print>
```

Keep in mind that you can only inspect variables in a Python shell. For example, save and run the following script:

```
phrase = "Hello, world"
phrase
```

The script executes without any errors, but no output is displayed! Throughout this book, you will see examples that use the interactive window to inspect variables.

Leave feedback on this section »

3.5 Leave Yourself Helpful Notes

Programmers often read code they wrote several months ago and wonder "What the heck does this do?" Even with descriptive variable names, it can be difficult to remember why you wrote something the way you did when you haven't looked at it for a long time.

To help avoid this problem, you can leave comments in your code. **Comments** are lines of text that don't affect the way the script runs. They help to document what's supposed to be happening.

In this section, you will learn three ways to leave comments in your code. You will also learn some conventions for formatting comments, as well as some pet peeves regarding their over-use.

How to Write a Comment

The most common way to write a comment is to begin a new line in your code with the `#` character. When your code is run, any lines starting with `#` are ignored. Comments that start on a new line are called **block comments**.

You can also write **in-line comments**, which are comments that appear on the same line as some code. Just put a `#` at the end of the line of code, followed by the text in your comment.

Here is an example of the `hello_world.py` script with both kinds of comments added in:

```
# This is a block comment.

phrase = "Hello, world."
print(phrase)  # This is an in-line comment.
```

The first line doesn't do anything, because it starts with a `#`. Likewise, `print(phrase)` is executed on the last line, but everything after the `#` is ignored.

Of course, you can still use the `#` symbol inside of a string. For instance, Python won't mistake the following for the start of a comment:

```
print("#1")
```

In general, it's a good idea to keep comments as short as possible, but sometimes you need to write more than will reasonably fit on a single line. In that case, you can continue your comment on a new line that also begins with a `#` symbol:

```
# This is my first script.
# It prints the phrase "Hello, world."
# The comments are longer than the script!

phrase = "Hello, world."
print(phrase)
```

Besides leaving yourself notes, comments can also be used to **comment out** code while you're testing a program. In other words, adding a `#` at the beginning of a line of code lets you run your program as if that line of code didn't exist without having to delete any code.

To comment out a section of code in IDLE highlight one ore more lines

to be commented and press:

- **Windows:** Alt + 3
- **macOS**: Ctrl + 3
- **Ubuntu Linux:**: Ctrl + D

Two # symbols are inserted at the beginning of each line. This doesn't follow PEP 8 comment formatting conventions, but it gets the job done!

To un-comment out your code and remove the # symbols from the beginning of each line, highlight the code that is commented out and press:

- **Windows:** Alt + 4
- **macOS:** Ctrl + 4
- **Ubuntu Linux:** Ctrl + Shift + D

Now let's look at some common conventions regarded code comments.

Conventions and Pet Peeves

According to PEP 8, comments should always be written in complete sentences with a single space between the # and the first word of the comment:

```
# This comment is formatted to PEP 8.

#don't do this
```

For in-line comments, PEP 8 recommends at least two spaces between the code and the # symbol:

```
phrase = "Hello, world"  # This comment is PEP 8 compliant.
print(phrase)# This comment isn't.
```

A major pet peeve among programmers are comments that describe

what is already obvious from reading the code. For example, the following comment is unnecessary:

```
# Print "Hello, world"
print("Hello, world")
```

No comment is needed in this example because the code itself explicitly describes what is being done. Comments are best used to clarify code that may not be easy to understand, or to explain why something is done a certain way.

In general, PEP 8 recommends that comments be used sparingly. Use comments only when they add value to your code by making it easier to understand *why* something is done a certain way. Comments that describe *what* something does can often be avoided by using more descriptive variable names.

Leave feedback on this section »

3.6 Summary and Additional Resources

In this chapter, you wrote and executed your first Python program! You wrote a small program that displays the text `"Hello, world"` using the `print()` function.

You were introduced to three concepts:

1. **Variables** give names to values in your code using the assignment operator (`=`)
2. **Errors**, such as syntax errors and run-time errors, are raised whenever Python can't execute your code. They are displayed in IDLE's interactive window in the form of a traceback.
3. **Comments** are lines of code that don't get executed and serve as documentation for yourself and other programmers that need to read your code.

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-3

Additional Resources

To learn more, check out the following resources:

- 11 Beginner Tips for Learning Python Programming
- Writing Comments in Python (Guide)
- Recommended resources on realpython.com

Leave feedback on this section »