Python Programming
Module 2: Variables And Simple Data Types

Learning objectives

1. Working with variables.
2. Descriptive variable names and how to resolve name errors and syntax errors when they arise.
3. What strings are and how to display strings using lowercase, uppercase, and titlecase.
4. Using whitespace to organize output neatly and striping unneeded whitespace from different parts of a string.
5. Start working with integers and floats. Learn about some unexpected behaviour to watch out for when working with numerical data.
6. Write explanatory comments to make your code easier for you and others to read.

What happens when you run hello_world.py

So, what Python does when you run `hello_world.py`. As it turns out, Python does a fair amount of work, even when it runs a simple program:

```
print("Hello Python world!")
```

When you run this code, you should see this output:

```
Hello Python world!
```

When you run the file `hello_world.py`, the ending `.py` indicates that the file is a Python program. Your editor then runs the file through the `Python interpreter`, which reads through the program and determines what each word in the program means. For example, when the interpreter sees the word `print`, it prints to the screen whatever is inside the parentheses.

As you write your programs, your editor highlights different parts of your program in different ways. For example, it recognizes that print is the name of a function and displays that word in blue. It recognizes that `“Hello Python world!”` is not Python code and displays that phrase in orange. This feature is called `syntax highlighting` and is quite useful as you start to write your own programs.

Variables

Let’s try using a variable in `hello_world.py`. Add a new line at the beginning of the file, and modify the second line:

```
message="Hello Python world!"
print(message)
```

You should see the same output you saw previously:

```
Hello Python world!
```

We’ve added a `variable` named `message`. Every variable holds a `value`, which is the information associated with that variable. In this case the value is the text `“Hello Python world!”.`
Adding a variable makes a little more work for the Python interpreter. When it processes the first line, it associates the text `“Hello Python world!”` with the variable message. When it reaches the second line, it prints the value associated with message to the screen.
Let’s expand on this program by modifying `hello_world.py` to print a second message. Add a blank line to `hello_world.py`, and then add two new lines of code:

```
message="Hello Python world!"
print(message)

message="Hello Python Crash Course world!"
print(message)
```

Now when you run `hello_world.py`, you should see two lines of output:

```
Hello Python world!
Hello Python Crash Course world!
```

You can change the value of a variable in your program at any time, and Python will always keep track of its current value.

Naming and Using Variables

When you're using variables in Python, you need to adhere to a few rules and guidelines. Breaking some of these rules will cause errors; other guidelines just help you write code that's easier to read and understand. Be sure to keep the following variable rules in mind:

- Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable `message_1` but not `1_message`.
- Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example, `greeting_message` works, but `greeting message` will cause errors.
- Do not use words that Python has reserved for a particular programmatic purpose, such as the word print. (See "Python Keywords and Built-in Functions" at the end of document.)
- Variable names should be short but descriptive. For example, `name` is better than `n`, `student_name` is better than `s_n`, and `name_length` is better than `length_of_persons_name`.
- Be careful when using the lowercase letter l and the upper case letter O because they could be confused with the numbers 1 and 0.

The Python variables you're using at this point should be lowercase. You won't get errors if you use uppercase letters, but it's a good idea to avoid using them for now.

Avoiding Name Errors When Using Variables

Let's look at an error you're likely to make early on and learn how to fix it. We'll write some code that generates an error on purpose. Enter the following code, including the misspelled word `mesage` shown in bold:

```
message="Hello Python Crash Course reader!"
print(mesage)
```

When an error occurs in your program, the Python interpreter does its best to help you figure out where the problem is. The interpreter provides a traceback when a program cannot run successfully. A traceback is a record of where the interpreter ran into trouble when trying to execute your code. Here's an example of the traceback that Python provides after you've accidentally misspelled a variable's name:

```
  Traceback (most recent call last):
1    File "hello_world.py", line 2, in <module>
2         print(mesage)
```

```
3 NameError: name 'mesage' is not defined
```

The output at `1`, reports that an error occurs in line 2 of the file `hello_world.py`. The interpreter shows this line to help us spot the error quickly in `2` and `3` tells us what kind of error it found. In this case it found a `name` error and reports that the variable being printed, `mesage`, has not been defined. Python can't identify the variable name provided. A name error usually means we either forgot to set a variable's value before using it, or we made a spelling mistake when entering the variable's name. The best way to understand new programming concepts is to try using them in your programs.

Strings

The first data type we'll look at is the string. A string is simply a series of characters. Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings like this:

```
"This is a string."
'This is also a string.'
```

This flexibility allows you to use quotes and apostrophes within your strings:

```
'I told my friend, "Python is my favourite language!"'
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strength is its diverse and supportive community."
```

Let's explore some of the ways you can use strings.

Changing Case in a String with Methods

One of the simplest tasks you can do with strings is change the case of the words in a string. Look at the following code, and try to determine what's happening:

```
name="ada lovelace"
print(name.title())
```

Save this file as `name.py`, and then run it. You should see this output:

```
Ada Lovelace
```

In this example, the lowercase string `"ada lovelace"` is stored in the variable name. The method `title()` appears after the variable in the `print()` statement. A method is an action that Python can perform on a piece of data. The dot `(.)` after name in `name.title()` tells Python to make the `title()` method act on the variable name. Every method is followed by a set of parentheses, because methods often need additional information to do their work. That information is provided inside the parentheses. The `title()` function doesn't need any additional information, so its parentheses are empty. `title()` displays each word in title case, where each word begins with a capital letter. You might want your program to recognize the input values `Ada`, `ADA`, and `ada` as the same name, and display all of them as `Ada`.

Several other useful methods are available for dealing with case as well. For example, you can change a string to all uppercase or all lowercase letters like this:

```
  name="Ada Lovelace"
  print(name.upper())
  print(name.lower())
```

This will display the following:

```
  ADA LOVELACE
  ada lovelace
```

The `lower()` method is particularly useful for storing data.

Combining or Concatenating Strings

It's often useful to combine strings. For example, you might want to store a first name and a last name in separate variables, and then combine them when you want to display someone's full name:

```
  first_name="ada"
  last_name="lovelace"
1 full_name=first_name+" "+last_name

  print(full_name)
```

Python uses the plus symbol (+) to combine strings. In this example, we use + to create a full name by combining a `first_name`, a space, and a `last_name` at `1`, giving this result:

```
  ada lovelace
```

This method of combining strings is called `concatenation`. You can use concatenation to compose complete messages using the information you've stored in a variable. Let's look at an example:

```
  first_name="ada"
  last_name="lovelace"
  full_name=first_name+" "+last_name

1 print("Hello, "+full_name.title()+"!")
```

Here, the full name at `1` is used in a sentence that greets the user, and the `title()` method is used to format the name appropriately. This code returns a simple but nicely formatted greeting:

```
  Hello, Ada Lovelace!
```

You can use concatenation to compose a message and then store the entire message in a variable:

```
  first_name="ada"
  last_name="lovelace"
  full_name=first_name+" "+last_name

1 message="Hello, "+full_name.title()+"!"
2 print(message)
```

This code displays the message “`Hello, Ada Lovelace!`” as well, but storing the message in a variable at `1` makes the final print statement at `2` much simpler.

Adding Whitespace to Strings with Tabs or Newlines

In programming, whitespace refers to any non printing character, such as spaces, tabs, and end-of-line symbols. You can use whitespace to organize your output so it's easier for users to read.
To add a tab to your text, use the character combination `\t` as shown at `1`:

```
  >>> print("Python")
  Python
1 >>> print("\tPython")
      Python
```

To add a newline in a string, use the character combination `\n`:

```
  >>> print("Languages:\nPython\nC\nJavaScript")
  Languages:
  Python
  C
  JavaScript
```

You can also combine tabs and newlines in a single string. The string `"\n\t"` tells Python to move to a new line, and start the next line with a tab. The following example shows how you can use a one-line string to generate four lines of output:

```
  >>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
  Languages:
      Python
      C
      JavaScript
```

Stripping Whitespace

To programmers `'python'` and `'python '` look pretty much the same. But to a program, they are two different strings. Python detects the extra space in `'python '` and considers it significant unless you tell it otherwise.
Python can look for extra whitespace on the right and left sides of a string. To ensure that no whitespace exists at the right end of a string, use the `rstrip()` method.

```
1 >>> favorite_language='python '
```

```
2 >>> favorite_language
  'python '
3 >>> favorite_language.rstrip()
  'python'
4 >>> favorite_language
  'python '
```

The value stored in `favorite_language` at `1` contains extra whitespace at the end of the string. When you ask Python for this value in a terminal session, you can see the space at the end of the value `2` When the `rstrip()` method acts on the variable `favorite_language` at `3`, this extra space is removed. However, it is only removed temporarily. If you ask for the value of `favorite_language` again, you can see that the string looks the same as when it was entered, including the extra whitespace `4`. To remove the whitespace from the string permanently, you have to store the stripped value back into the variable:

```
  >>> favorite_language='python '
1 >>> favorite_language=favorite_language.rstrip()
  >>> favorite_language
  'python'
```

To remove the whitespace from the string, you strip the whitespace from the right side of the string and then store that value back in the original variable, as shown at `1`.
You can also strip whitespace from the left side of a string using the `lstrip()` method or strip whitespace from both sides at once using `strip()`:

```
1 >>> favorite_language=' python '
2 >>> favorite_language.rstrip()
  ' python'
3 >>> favorite_language.lstrip()
  'python '
4 >>> favorite_language.strip()
  'python'
```

In this example, we start with a value that has whitespace at the beginning and the end `1`. We then remove the extra space from the right side at `2`, from the left side at `3`, and from both sides at `4`. Experimenting with these stripping functions can help you become familiar with manipulating strings. In the real world, these stripping functions are used most often to clean up user input before it's stored in a program.

Avoiding Syntax Errors with Strings

One kind of error that you might see with some regularity is a syntax error. A `syntax error` occurs when Python doesn't recognize a section of your program as valid Python code. For example, if you use an apostrophe within single quotes, you'll produce an error. This happens because Python interprets everything between the first single quote and the apostrophe as a string. It then tries to interpret the rest of the text as Python code, which causes errors.

Here's how to use single and double quotes correctly. Save this program as `apostrophe.py` and then run it:

```
message="One of Python's strengths is its diverse community."
print(message)
```

The apostrophe appears inside a set of double quotes, so the Python interpreter has no trouble reading the string correctly:

```
One of Python's strengths is its diverse community.
```

However, if you use single quotes, Python can't identify where the string should end:

```
message='One of Python's strengths is its diverse community.'
print(message)
```

You'll see the following output:

```
File "apostrophe.py", line 1
     message='One of Python's strengths is its diverse community.'
                           ^1
SyntaxError: invalid syntax
```

In the output you can see that the error occurs at `1` right after the second single quote. This `syntax error` indicates that the interpreter doesn't recognize something in the code as valid Python code. Errors can come from a variety of sources, and I'll point out some common ones as they arise. You might see syntax errors often as you learn to write proper Python code. Syntax errors are also the least specific kind of error, so they can be difficult and frustrating to identify and correct.

Your editor's syntax highlighting feature should help you spot some syntax errors quickly as you write your programs. If you see Python code highlighted as if it's English or English highlighted as if it's Python code, you probably have a mismatched quotation mark somewhere in your file.

Printing in Python 2

The print statement has a slightly different syntax in Python 2:

```
>>> python2.7
>>> print "Hello Python 2.7 world!"
Hello Python 2.7 world!
```

Parentheses are not needed around the phrase you want to print in Python 2. Technically, print is a function in Python 3, which is why it needs parentheses. Some Python 2 print statements do include parentheses, but the behaviour can be a little different than what you'll see in Python 3. Basically, when you're looking at code written in Python 2, expect to see some print statements with parentheses and some without.

Numbers

Numbers are used quite often in programming to keep score in games, represent data in visualizations, store information in web applications, and so on. Python treats numbers in several different ways, depending on how they are being used. Let's first look at how Python manages integers, because they are the simplest to work with.

Integers

You can `add (+), subtract (-), multiply (*), and divide (/)` integers in Python.

```
>>> 2+3
5
>>> 3-2
1
>>> 2*3
6
>>> 3/2
1.5
```

In a terminal session, Python simply returns the result of the operation. Python uses two multiplication symbols to represent exponents:

```
>>> 3**2
9
>>> 3**3
27
>>> 10**6
1000000
```

Python supports the order of operations too, so you can use multiple operations in one expression. You can also use parentheses to modify the order of operations so Python can evaluate your expression in the order you specify. For example:

```
>>> 2+3*4
14
>>> (2+3)*4
20
```

The spacing in these examples has no effect on how Python evaluates the expressions; it simply helps you more quickly spot the operations that have priority when you're reading through the code.

Floats

Python calls any number with a decimal point a `float`. This term is used in most programming languages, and it refers to the fact that a decimal point can appear at any position in a number. Every programming language must be carefully designed to properly manage decimal numbers so numbers behave appropriately no matter where the decimal point appears.

For the most part, you can use decimals without worrying about how they behave. Simply enter the numbers you want to use, and Python will most likely do what you expect:

```
>>> 0.1+0.1
0.2
>>> 0.2+0.2
0.4
>>> 2*0.1
0.2
>>> 2*0.2
20.4
```

But be aware that you can sometimes get an arbitrary number of decimal places in your answer:

```
>>> 0.2+0.1
0.30000000000000004
>>> 3*0.1
0.30000000000000004
```

This happens in all languages and is of little concern. Python tries to find a way to represent the result as precisely as possible, which is sometimes difficult given how computers have to represent numbers internally. Just ignore the extra decimal places for now; you'll learn ways to deal with the extra places when you need to in the projects in Part II.

Avoiding Type Errors with the `str()` Function

Often, you'll want to use a variable's value within a message. For example, say you want to wish someone a happy birthday. You might write code like this:

```
age=23
message="Happy "+age+"rd Birthday!"

print(message)
```

You might expect this code to print the simple birthday greeting, `Happy 23rd birthday!` But if you run this code, you'll see that it generates an error:

```
Traceback (most recent call last):
    File "birthday.py", line 2, in <module>
         message="Happy "+age+"rd Birthday!"
```

```
1 TypeError: Can't convert 'int' object to str implicitly
```

This is a `type error`. It means Python can't recognize the kind of information you're using. In this example Python sees at `1` that you're using a variable that has an integer value `(int)`, but it's not sure how to interpret that value. Python knows that the variable could represent either the numerical value 23 or the characters 2 and 3. When you use integers within strings like this, you need to specify explicitly that you want Python to use the integer as a string of characters. You can do this by wrapping the variable in the `str()` function, which tells Python to represent non-string values as strings:

```
age=23
message="Happy "+str(age)+"rd Birthday!"

print(message)
```

Python now knows that you want to convert the numerical value 23 to a string and display the characters 2 and 3 as part of the birthday message. Now you get the message you were expecting, without any errors:

```
Happy 23rd Birthday!
```

Working with numbers in Python is straightforward most of the time. If you're getting unexpected results, check whether Python is interpreting your numbers the way you want it to, either as a numerical value or as a string value.

Integers in Python 2

Python 2 returns a slightly different result when you divide two integers:

```
>>> python2.7
>>> 3/2
1
```

Instead of `1.5`, Python returns `1`. Division of integers in Python 2 results in an integer with the remainder truncated. Note that the result is not a rounded integer; the remainder is simply omitted.
To avoid this behaviour in Python 2, make sure that at least one of the numbers is a float. By doing so, the result will be a float as well:

```
>>> 3/2
1
>>> 3.0/2
1.5
>>> 3/2.0
1.5
>>> 3.0/2.0
1.5
```

This division behavior is a common source of confusion when people who are used to Python 3 start using Python 2, or vice versa. If you use or create code that mixes integers and floats, watch out for irregular behavior.

Comments

Comments are an extremely useful feature in most programming languages. Everything you've written in your programs so far is Python code. As your programs become longer and more complicated, you should add notes within your programs that describe your overall approach to the problem you're solving. A `'comment'` allows you to write notes in English within your programs.

How Do You Write Comments?

In Python, the hash mark (`#`) indicates a comment. Anything following a hash mark in your code is ignored by the Python interpreter. For example:

```
#Say hello to everyone.
print("Hello Python people!")
```

Python ignores the first line and executes the second line.

```
Hello Python people!
```

What Kind of Comments Should You Write?

The main reason to write comments is to explain what your code is supposed to do and how you are making it work. When you're in the middle of working on a project, you understand how all of the pieces fit together. But when you return to a project after some time away, you'll likely have forgotten some of the details. You can always study your code for a while and figure out how segments were supposed to work, but writing good comments can save you time by summarizing your overall approach in clear English.

Exercises:

1. **Write a separate program to perform each of these exercises.**
    a. Store a message in a variable, and then print that message.
    b. Store a message in a variable, and print that message. Then change the value of your variable to a new message, and print the new message.
    c. Write addition, subtraction, multiplication, and division operations that each result in the number 8. Your output should simply be four lines with the number 8 appearing once on each line as result.
    d. Store 10 in a variable. Then, using that variable, create a message that says `'10 is my lucky number'`. Print that message.
    e. Store `'Julia'` in a variable, and print a message saying: `"Hello Julia, would you like to learn some Python today?"`.
    f. Store a your name in a variable, and then print that name in lowercase, uppercase, and titlecase.

2. **Store your name, and include whitespace characters at the beginning and end of the name. Make sure you use each character combination, `"\t"` and `"\n"`, at least once, and**
    a. Print the name once, so the whitespace around the name is displayed.
    b. Print the name using each of the three stripping functions, `lstrip()`, `rstrip()`, and `strip()`.

3. **Calculate the length of the hypotenuse of a right-angled triangle where the other sides have lengths 3 & 4.**
    a. Create variables called `"b"` and `"c"` and assign them the values of the sides with known lengths.
    b. Write a mathematical express to calculate the length of the hypotenuse (Hint: $a^2 = b^2 + c^2$).
    c. Create a variable `"a"` that equals the result of the calculation.
    d. Print the value of variable `"a"`.

4. **Write a program to convert 37.5 degree celsius to degree fahrenheit.**

5. **Write a Python program to print the following string in a specific format (see the output) only using escape sequences (i.e. `\t`, `\n`, ...).**
    Sample String : "Twinkle, twinkle, little star, How I wonder what you are! Up above the world so high, Like a diamond in the sky. Twinkle, twinkle, little star, How I wonder what you are".

    Output :
```
Twinkle, twinkle, little star,
	How I wonder what you are!
		Up above the world so high,
		Like a diamond in the sky.
Twinkle, twinkle, little star,
How I wonder what you are.
```

6. **Write a Python program to get a string made of the first 2 and the last 2 chars from a given string. If the string length is less than 2, return empty string.**
    a. Sample String : `'pytho'`
       Expected Result : `'pyho'`

    b. Sample String : `'py'`
       Expected Result : `'pypy'`

    c. Sample String : `'p'`
       Expected Result : Empty String

7. **Write a Python program to get a single string from two different strings, separated by a space and swap the first two characters of each string.**
   Input : 'abc', 'xyz'
   Output: 'xyc abz'

8. **Write a Python program to change a given string to a new string where the first and last characters have been exchanged.**