ICS3U: Input and Output

The Code Window

So far, the shell has been adequate for performing simple mathematical calculations; however, it is not very efficient for entering larger programs. The **code window** is a better option, providing us with an area in which we can enter an arbitrary amount of code. It also allows a programmer to save programs, and to run them or modify them at later dates.

One thing to note about the code window is that it does not echo values like the shell does. Entering the command `3+5` in the code window will produce no **output** when it is run. To see the results of such calculations, we will need to use some additional functions.

Displaying Output

To output one or more values to the screen, use the built-in `print` function. The general format of `print` is something like this:

```
print(objects, keyword options)
```

In its most basic form, `print` takes one or more values as **arguments**. The statement `print(3, 4, 5)` will display the following in the shell.

```
3 4 5
```

Note that values must be separated by commas inside of `print`, and that values are separated by single spaces in the output. Any object (a value, variable, function, and so on) can be output via `print`. Try running the code below to see the output.

```
number = 10
animal = "dog"
print("My favourite number is", number, "and my favourite animal is a", animal)
```

Four arguments are present in the `print` statement: a literal string, a variable of type `int`, a second literal string, and a variable of type `str`. All are output on a single line, separated by spaces. While the literal strings were enclosed in double quotes, variables are not. Students sometimes think that all values output via `print` must have quotes, but the following code should demonstrate why this is false.

```
number = 10
print(number)
print("number")
```

To change some of the default behaviours of print, you may specify some **keyword options** after the values. All keyword options follow the format `option_name = value`. In the last example, the two `print` statements resulted in two separate lines of output. We can change the end-of-line behaviour using the `end` option.

```
print("All of this text ", end="")
print("is on the same line.")
```

In this case, the new end-of-line character is an empty string, so both statements are output on the same line. This can be changed to any character, or sequence of characters, if desired.

```
print("This line ends with two dollar signs and a newline.", end="$$\n")
print("This is on separate line.")
```

In the example above, newline is represented as `\n`. This is called an **escape character**. They are often used to represent special whitespace characters such as newlines, carriage returns (`\r`), and tabs (`\t`), but they can also be used to represent control sequences such as backspace (`\b`). They can also be used to display certain characters that would normally not be displayed in strings, such as quotes, double quotes, and backslashes. Try the following `print` statements to see how they are replaced.

```
print("Use a backslash, \\, as part of an escape character.")
print("Tom bumped his arm on a rock. \"Ow!\" he cried out.")
print("'I love Hallowe\'en,' he said.")
```

Another keyword option is `sep`, which can alter the character that separates values. To suppress spacing between values, use an empty string.

```
print("Ca", "na", "da", sep="")
```

This can be useful when several values need to be separated by a specific character, like a comma.

```
x=3
y=5
z=7
print(x, y, z, sep=",")
```

It can also be useful if you want to eliminate the pesky space that is inserted between a variable and a period at the end of a sentence.

```
number = 2
print("A really nice number is ", number, ".", sep="")
```

Typecasting

We mentioned earlier that Python is a dynamically-typed language, and that data types can change as a result of a calculation. Sometimes we want to change a variable's data type manually. For example, we have seen how concatenating an integer with a string results in a run-time error.

```
>>> "hello" + 3
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Any numeric data type can be converted to a string using the `str` function.

```
>>> "hello" + str(3)
'hello3'
```

This explicit conversion from one data type to another is called **typecasting**. Conversions to other data types can be achieved using additional functions such as `int`, `float`, `list` and so on.

Only certain strings can be converted to numeric types. Here are some examples that work.

```
>>> int("7")
7
>>> int("-9")
-9
>>> int(8.5)
8
>>> float("7")
7.0
>>> float(7)
7.0
```

Note that a `float` will lose all digits after the decimal point if it is converted to an `int`, as is the case in the third example. Any `int` can be converted to a `float`. Here are some examples that do not work.

```
>>> int("123abc")
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '123abc'
>>> int("8.5")
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '8.5'
>>> int("7.0")
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '7.0'
```

The first conversion fails because there are non-numeric characters in the string. The second is unsuccessful because 8.5 cannot be represented as an integer. Note that this is *different* behaviour than converting from a float value of 8.5 to an integral value of 8. As we saw above, this “works” by truncating all values after the decimal point. The third example is similar, but may be surprising, since we as humans understand that 7.0 represents the same value as 7. The Python interpreter, on the other hand, does not.

Obtaining User Input

Computers would not be nearly as useful as they are if they were unable to obtain and process user input. Python can do this using the `input` function. The general format is something like this:

```
variable = input(prompt)
```

When you run a program and it encounters a call to the `input` function, the **prompt** will be displayed in the shell, followed by a cursor. The program will wait for the user to enter something, followed by the Enter (Return) key, before moving on to the next command. User input is assigned to the specified variable. Try the following code, entering your name when prompted to do so.

```
name = input("What is your name? ")
print("Hello ", name, "!", sep="")
```

All input is interpreted as a string. This can cause a problem when attempting to perform mathematical operations on user input. Below is code that attempts to double a user-entered number.

```
num = input("Enter a number: ")
double = num * 2
print(double, "is twice your number.")
```

The output, however, looks like this:

```
Enter a number: 10
1010 is twice your number.
```

Obviously, there is a logical error in the code. In this case, the error occurs on lie second line. This is because the value stored in `num` is a string. When the multiplication operator is used with a string and an integer, the string is repeated that many times, as in the following:

```
>>> "ha"*3
'hahaha'
```

To fix things, we can typecast num as an `int`.

```
num = input("Enter a number: ")
double = int(num) * 2
print(double, "is twice your number.")
```

This will produce the expected result.

```
Enter a number: 10
20 is twice your number.
```

A common approach is to typecast user input as it is read. The same result could have been achieved via the following code, which would immediately assign an integral value to `num`.

```
num = int(input("Enter a number: "))
double = num * 2
print(double, "is twice your number.")
```

One important difference between the format of the `print` and `input` functions is that `print` allows for multiple arguments, whereas `input` can only use a single string for its prompt. Attempting to use multiple arguments in an `input` function causes a run-time error.

```
num1 = int(input("Enter an integer: "))
num2 = int(input("Enter an integer larger than ", num1, ": "))

Traceback (most recent call last):
  File "/home/Code/user_input.py", line 2, in <module>
    num2 = int(input("Enter an integer larger than ", num1, ": "))
TypeError: input expected at most 1 argument, got 3
```

One way to get around this limitation is to build a string first using concatenation, then use it as the prompt in the `input` function.

```
num1 = int(input("Enter an integer: "))
prompt = "Enter an integer larger than " + str(num1) + ": "
num2 = int(input(prompt))
```

Remember that `num1` needs to be typecast as a string, or we will receive a run-time error when attempting to concatenate a string with an integer.