ICS3U: Data Types and Variables

Data Types

Data come in different forms. As humans, we recognize the value 3 as a number, and the value "Apple" as a sequence of letters. We can manipulate some data using mathematical operations, such as adding 3 and 5 to get 8; however, it doesn't make sense to ask someone to calculate the value of 3 + "Apple". This is because the data 3 and "Python" are different **types**.

Like humans, Python treats different values in different ways, depending on their types. To check a value's type, Python provides the built-in `type` function.

```
>>> type(3)
<class 'int'>
```

Since 3 is an integer value, Python identifies it as an `int` type. There are three **numeric data types** in Python: **integers** (`int`); **floating point values** (`float`), or decimal values; and **complex numbers** (`complex`). We do not cover complex numbers in this course, so you can ignore them for now.

```
>>> type(3.14159)
<class 'float'>
>>> type(5+3j)
<class 'complex'>
```

In addition to numeric data types, Python also has **sequence data types**. The most common of these are **strings** (`str`), which are sequences of letters, numbers, whitespace, special characters, and so on. Strings are always urrounded by either single or double quotes – the choice is yours, but you must be consistent. Other sequences, including **tuples** (`tuple`), **lists** (`list`) and **dictionaries** (`dict`), will be covered in more detail later in this course.

```
>>> type("Apple")
<class 'str'>
>>> type((3, 5))
<class 'tuple'>
>>> type([3, 7, 4])
<class 'list'>
```

Certain mathematical operators only work with certain data types. Operations like addition and multiplication, for instance, will work with any two numeric data types.

```
>>> type(3)
<class 'int'>
>>> type(8.5)
<class 'float'>
>>> 3 + 8.5
11.5
>>> type(11.5)
<class 'float'>
```

You may have noticed earlier that division using `/` always returns a float, even when the result is an integral value. To force an integer result, use `//` instead. Note that this will **truncate** (cut off) any decimals that follow the integer.

```
>>> 20 / 3
6.666666666666667
>>> 20 // 3
6
```

As we mentioned earlier, it does not make any sense to "add" a string and an integer. A run-time error will occur if you try.

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

Since the first value encountered by the interpreter is a string, the error occurs when trying to **concatenate** (join together) the string and an integer. Concatenation can only be done with strings, as in the following example.

```
>>> "he" + "llo"
'hello'
```

If the order is swapped, a different run-time error will occur.

```
>>> 3 + "hello"
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

This time, the interpreter encounters an integer first. It attempts to add the string "hello" to the integer value of 3, which does not make sense mathematically, so an error is generated instead.

Variables

In mathematics, we often use **variables** to represent an unknown value, or to represent a value for which an arbitrary value can be assigned. Python also uses variables, as do virtually all programming languages, in the second sense. In Python, a variable is a symbolic name that is attached to an **object**. An object in Python is a value, a sequence of values, or a combination of values and functions (called a **class**). In fact, nearly everything in Python is an object in some form.

Variables are generally used for two main purposes: to store a value (or sequence of values) using a name that conveys meaning to the programmer, or to provide a mechanism where a value can change when necessary. For example, a physicist that regularly uses the value of the acceleration due to gravity at sea level (9.81 m/s^2) might create a variable, *g*, and assign this value to it. This makes it possible for the programmer to do calculations by referring to *g*, rather than having to type out its **literal value**.

A variable is created when a value is assigned to it. Python uses the **assignment operator**, =, to do this. Space in main memory is allocated to store the value, and a link is made between the variable name and the location in memory.

```
>>> g = 9.81
```

The variable's name is always on the left, and the value to be assigned is always on the right. Swapping them will cause a syntax error. We can now refer to the variable by its name, anytime we want. In the shell, we can display (or **echo**) the value of a variable by typing its name.

```
>>> g
9.81
```

Mathematical operations can be performed on a variable, just as if it was a literal value, assuming it is a numeric type. To double the value of *g*, for example, we can do the following.

```
>>> g * 2
19.62
>>> g
9.81
```

Note that this does not change the value of *g* itself. The interpreter is calculating a value twice that of *g*, but a numeric variable will not change its value unless it is explicitly reassigned a new one. This can be done using something like the following.

```
>>> g = g * 2
>>> g
19.62
```

In mathematics, variables often have short one- or two-letter names like *x* or *y*; however, most programming languages variables allow us to use longer, more descriptive names that describe what they represent. It is good practice to avoid single-letter variables, unless there is a well-known association between the latter and a concept. For example, a program that uses two values to represent a coordinate (*x*, *y*) might simply use *x* and *y* for variables. However, while the following two lines are syntactically the same, the second is *much* easier to understand from a human perspective.

```
>>> a = b * c
>>> distance = rate * time
```

The following are all "good" variable names.

```
>>> age = 16
>>> price = 13.49
>>> salesTaxRate = 0.13
>>> number_of_players = 4
```

The last two examples show different ways to name variables involving multiple words. The first, `salesTaxRate`, is written in **Camel Case**, where words are delimited using capital letters. The second, `number_of_players`, is written in **Snake Case**, where words are separated by underscores. The choice of which to use is up to you, but you should try to be consistent. The official Python guidelines recommend the use of Snake Case, citing improved readability.

Variable names can contain letters (uppercase or lowercase), numbers (but not as the first character) and underscores only. Additionally, a variable can not have the same name as one of Python's keywords, which are listed below. This would interfere with the interpreter's ability to process code.

```
and         as          assert      async       await       break       class
continue    def         del         elif        else        except      False
finally     for         from        global      if          import      in
is          lambda      nonlocal    None        not         or          pass
raise       return      True        try         while       with        yield
```

Here are some examples of unacceptable variable names.

```
>>> 1st_name="Jon"
SyntaxError: invalid syntax
>>> number of people = 10
SyntaxError: invalid syntax
>>> class = 7
SyntaxError: invalid syntax
```

Assignment Operators

Earlier, we changed the value of *g* from 9.81 to 19.62 using the following command.

```
>>> g = g * 2
```

While we probably don't want to change the acceleration due to gravity in real life, there are many instances in programming where we want to change a variable's current value to something else. Python provides a number of **assignment operators** that provide us with a short-hand notation for doing exactly this. The following command is equivalent to the one above.

```
>>> g *= 2
```

Assignment operators exist for all mathematical operations, including addition, subtraction, multiplication, division, exponentiation, quotient, and remainder. The format is the same for each of them. So to add 1 to the value stored in age, we could type the command below.

```
>>> age += 1
```

Since variables act as literal values, it is possible to use an assignment operator entirely with variables.

```
>>> total -= amount
```

It is important to note that an assignment operator will not create a new variable, but only modify an existing one. If you try to use an assignment operator with a variable that has not yet been created, a run-time error will occur.

```
>>> ThisVariableDoesNotExist += 1
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
    ThisVariableDoesNotExist += 1
NameError: name 'ThisVariableDoesNotExist' is not defined
```