ICS4U: REVIEW OF PYTHON

1. PYTHON "BASICS"

Python is a powerful interpreted language that is easy to learn and capable of doing complex things. Before it is possible to move on to more sophisticated programs, it is necessary to review how to perform basic actions, such as manipulating values, and display the results of calculations.

1.1. Mathematics

Mathematics in Python uses the standard symbols for addition, subtraction, multiplication and division: `+`, `-`, `*` and `/` respectively. Exponentiation uses `**`, not `^` which is a bitwise operator. Integer division (quotient) is obtained using `//`, and modulus (remainder) with `%`.

```
>>> 6+4
10
>>> 8/5
1.6
>>> 8//5
1
>>> 8%5
3
```

Mathematical operations follows the traditional order of operations (PEMDAS, BEDMAS). Round brackets can be used to group operations if necessary.

```
>>> 3+2*5
13
>>> (3+2)*5
25
>>> 4**2-15//5
13
```

Some additional mathematical functions are available as built-in functions. These include `abs` (absolute value), `min` and `max` (for finding the smallest and largest elements in a sequence), `sum` (for finding the sum of a sequence composed of numeric elements), and `round` (for rounding to a specified number of digits). Both `max` and `min` take any number of **arguments**, while `abs` takes only one, and `round` takes two.

```
>>> min(3, 1, 7)
1
>>> round(3.785, 1)
3.8
```

Additional mathematical functions (and constants) can be accessed by importing the standard `math` module that ships with Python. As with all modules, it is necessary to prefix the name of the function or constant with the name of the module.

```
>>> import math
>>> math.sqrt(9)
3.0
>>> math.pi
3.141592653589793
```

A full list of functions, including their arguments and returned values, can be found in the official documentation. Commonly-used functions include `sqrt` (square root), `factorial`, `log`, and the trigonometric functions `sin`, `cos` and `tan`. Note that the latter three operate in radians, not degrees.

1.2. Data Types

Python supports several built-in **data types**: numeric types, such as integers (`int`), floating-point values (`float`) and complex numbers (`complex`); sequences, such as strings (`str`), tuples (`tuple`), and lists (`list`); Booleans (`bool`); as well as others that handle sets, maps, binary sequences, etc.

A value's data type can be checked using `type`.

```
>>> type(3)
<class 'int'>
>>> type(4.5)
<class 'float'>
```

Some functions work only with certain data types. Others produce different output for different types.

1.3. Variables

Similar to in mathematical expressions, a **variable** is a reference to a value. Variables do not need to be declared before they are assigned a value using the **assignment operator**, `=`.

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

Variable names can consist of lowercase and capital letters, numbers (provided they are not the first character), and underscores. No spaces or other special characters are permitted. Since spaces are not allowed, variable names made up of multiple words typically use **camel case**, such as `numberOfCars`, or **snake case**, such as `number_of_cars`. To encourage readability and general understanding of a piece of code, variable names should be descriptive, unless they have an inherent mathematical association (*x* and *y* for coordinates, *r* for radius, etc.) or programming convention (*i*, *j*, *k* for loop counters) associated with them.

A variable name cannot be that of a Python reserved keyword, such as `class`. While it is possible to use the name of a built-in function as a variable name, this should be avoided. Doing so redefines the reference from the function to the variable and can lead to errors, as in this example.

```
>>> print = int(input("How many pages do you want to print? "))
How many pages do you want to print? 10
>>> print("Thank you.")
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'int' object is not callable
>>> print
10
```

1.4. Assignment Operators

It is common for computer programs to modify the values stored in variables as a program executes. Rather than create new variables each time a change is made, it is often sufficient to discard the previous value of a variable in favour of a new one, such as in the example below.

```
>>> x = 9
>>> x = x*2
>>> x
18
```

The command `x = x*2` doubles the value of *x*, then overwrites the existing value with the new one. This 'reassignment' happens so frequently that Python provides a set of **assignment operators** as a form of short-hand.

```
>>> x = 9
>>> x *= 2
>>> x
18
```

All of the basic mathematical operators have corresponding assignment operators. Both forms are syntactically correct, and usage is a matter of preference. The specific cases of addition (`+=`) or subtraction (`-=`) are often referred to as the **increment operator** and **decrement operator**.

1.5. Output and User Input

Output is handled by the `print` function.

```
>>> x = 5
>>> y = 3
>>> print("The product of", x, "and", y, "is", x*y)
The product of 5 and 3 is 15
```

Multiple arguments are separated by commas. Note that it is possible to include calculations, or function calls, as arguments. By default, Python inserts a single space between output terms. To change this, modify the value of the separator character using the `sep` keyword.

```
>>> print("Hello", "there", sep="**")
Hello**there
>>> print("Hello", "there", sep="")
Hellothere
```

Similarly, the `end` keyword will change the end-of-line character from the default newline (`\n`). Use an empty string or space to ensure that output is displayed on the same line.

User input is handled via the appropriately-named `input` function. In order to retain any values entered by the user, it is necessary to assign them to variables.

```
>>> s = input("Enter something: ")
Enter something: hello
>>> s
'hello'
```

Unlike `print`, `input` will only accept a single string argument. It is possible to create a string beforehand, using either **concatenation** (`+`) or another method, if necessary.

```
>>> first_name = input("What is your first name? ")
What is your first name? Jon
>>> prompt = "What is your last name, " + first_name + "? "
>>> last_name = int(input(prompt))
What is your last name, Jon?
```

1.6. Typecasting

User input is always represented as a string, even if only numbers are entered. It is necessary to convert, or **typecast**, input to an appropriate form using a built-in function like `int`, `float`, `str`, etc.

```
>>> n = int(input("Enter an integer: "))
Enter an integer: 3
>>> m = n*4
>>> print(m)
12
```

Failing to convert a value to a suitable type may result in various errors. In the code below, *n* is a string, so the `*` operator repeats the string four times.

```
>>> n = input("Enter an integer: ")
Enter an integer: 3
>>> m = n*4
>>> print(m)
'3333'
```

If a value cannot be represented as the specified type, a run-time error will occur. For example, entering a non-numeric value in place of an integer will cause the program to halt. Dealing with these issues is covered later.

```
>>> n = int(input("Enter an integer: "))
Enter an integer: hello
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hello'
```

1.7. Random Values

The `random` module contains functions for generating pseudo-random values, or making selections from a collection of objects. Commonly-used functions include `randint` and `randrange` for generating random integers on an interval, and `choice` for selecting a random value from a sequence.

```
>>> import random
>>> random.randint(1, 10)
8
>>> random.randrange(1, 10, 2)
3
>>> random.choice("abcde")
'b'
```

Note that `randrange` accepts a third (optional) argument, *step*, that acts as a “count by” value. Additionally, `randint(a, b)` will return an integer between *a* and *b* inclusive, but `randrange(a, b)` will not include *b* as a possible value. As always, a full list of functions is available in the official documentation.

1.8. Types of Errors

Errors are a regular occurrence when programming, especially with large or complex programs. Becoming adept at interpreting error messages, and finding ways to correct non-functioning code, is essential to becoming a good programmer. In general, errors fall into one of three categories.

1. A **syntax error** is when there is something with the way in which a command, function, or operation is written, such that it prevents a program from running. Typical syntax errors in Python include missing parentheses, quotes or commas; missing or extraneous indentation; and incorrect usage of mathematical operators.
   ```
   >>> x = 5
   >>> print("The value of x is" x)
     File "<pyshell>", line 1
       print("The value of x is" x)
                                 ^
   SyntaxError: invalid syntax
   ```

2. A **run-time error** occurs when a program executes. That is, the syntax is correct, but there is some other problem with a command or value that is not detected until the program is running. Run-time errors may result from attempting to use a variable that has not been assigned a value, passing an incorrect number of arguments to a function, applying a mathematical operation between two non-compatible types, and many other reasons.

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

3. A **logical error** is when there is a problem with the way in which the program was designed. They are particularly nasty, because the program will run, but may produce incorrect output or result in an unexpected action. Sometimes logical errors are easy to identify, but in other cases, they are much more difficult to identify and fix. Most IDEs ship with a built-in **debugger**, which can be a powerful tool to help monitor the values and states of objects as a program executes. Becoming proficient with the debugger can help make correcting logical errors easier.

```
>>> x = 3
>>> y = 5
>>> print("The average of", x, "and", y, "is", x+y/2)
The average of 3 and 5 is 5.5
```

In the example above, `x+y` should be inside of parentheses.

```
>>> print("The average of", x, "and", y, "is", (x+y)/2)
The average of 3 and 5 is 4.0
```

2. CONDITIONAL PROCESSING

While some programs simply execute one command after another in a predefined sequence, the majority make decisions based on calculated values or user input. In order to execute certain code in one case, and alternate code in another, a program must be structured based on conditions.

2.1. Boolean Values

A Boolean data type can take on two possible values: `True` or `False`. Both of these are reserved keywords in Python. A variable can be assigned a value of `True` or `False` just like any other value.

```
>>> play_again = True
>>> type(play_again)
<class 'bool'>
```

Booleans are typically used for keeping track of the state of a system (`is_running = True`), or for indicating whether or not a certain condtion has been met (`fully_factored = False`). In the latter case, the variable is often called a **flag**. It can be checked to see whether it is set to `True` or `False`.

2.2. Relational Operators

Relational operators, sometimes called **comparison operators**, are those that compare two values and to determine a relationship between them. Python includes six main relational operators: equality (`==`), inequality (`!=`), strictly greater than (`>`), strictly less than (`<`), greater than or equal to (`>=`) and less than or equal to (`<=`). There are also operators for checking membership (`in`) and object identity (`is`). Each comparison evaluates as a Boolean value, either `True` or `False`.

```
>>> 10 > 6
True
>>> 5 == 7
False
>>> 12 <= 12
True
```

2.3. Making Decisions

In order to make a decision, it is necessary compare two or more values. Often, this will involve checking the value of a literal or a variable. These comparisons are made inside of an "`if` block" involving three keywords: `if`, `elif` and `else`. `if` specifies the primary comparison, while `elif` specifies an alternate, mutually exclusive condition. `else` handles all cases that are not covered by any `if` or `elif` statements. For example, the code below checks whether a user-entered integer is positive, negative, or neither.

```python
num = int(input("Enter an integer: "))
if num > 0:
    print("Your integer is positive.")
elif num < 0:
    print("Your integer is negative.")
else:
    print("Your integer is neither positive nor negative.")
```

Python uses indentation to group code inside of an `if` block. Each condition is either `True` or `False`, and an `if` or `elif` statement will execute the code inside of it only if the result is `True`. If all conditions for `if` or `elif` statements are `False`, then the `else` statement will execute its code.

While there may be any number of `elif` statements (including none), there is always a single `if` statement that precedes them. The `else` statement is also optional, and there can be at most one instance. It must occur after all `elif` statements.

2.4. Nested `if` Statements

If multiple conditions need to be tested, it is possible to nest an `if` block inside of another using a second level of indentation.

```python
n = int(input("Enter an integer between 10 and 20: "))
if n >= 10:
    if n <= 20:
        print("Integer is between 10 and 20.")
    else:
        print("Integer is not in the range 10-20.")
else:
    print("Integer is not in the range 10-20.")
```

The inner `if` block will only execute if the condition for the outer `if` block is `True`. This is one way to enforce multiple conditions, or to organize cases.

It is possible to do further nesting, each requiring an additional level of indentation, but this may be difficult to follow. Typically, for situations involving several nested `if` blocks, it is preferrable to use logical operators instead.

2.5. Logical Operators

There are three **logical operators** in Python: `and`, `or` and `not`. Both `and` and `or` compare two Boolean values, *x* and *y*. If *x* and *y* are both `True`, then `x and y` is `True`; otherwise, `x and y` is `False`. If *x* and *y* are both False, then `x or y` is `False`; otherwise, `x or y` is `True`. Unlike `and` and `or`, `not` operates on a single Boolean value, changing it from `True` to `False` or *vice versa*.

```
>>> x = True
>>> y = False
>>> x and y
False
>>> x or y
True
>>> not x
False
```

Logical operators are typically used to group related conditions. Using `and` can often eliminate nested `if` blocks.

```python
n = int(input("Enter an integer between 10 and 20: "))
if n >= 10 and n <= 20:
    print("Integer is between 10 and 20.")
else:
    print("Integer is not in the range 10-20.")
```

Using `or` can reduce the number of, or eliminate, `elif` statements in an `if` block.

```python
import random
letter = random.choice("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
print(letter)
if letter == "C" or letter == "A" or letter == "T":
    print("The random letter is in 'CAT'.")
```

2.6. Implicit Truth Testing

A common logical error is to omit the comparisons when using logical operators.

```python
import random
letter = random.choice("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
print(letter)
if letter == "C" or "A" or "T":
    print("The random letter is in 'CAT'.")
```

Running this code will always result in the ouput `The random letter is in 'CAT'`. This is because when a condition is not *explicitly* stated, as in `letter == "C"`, Python associates an *implicit* truth value with each value or variable. All values that are `False`, 0, or empty sequences are implicitly `False`. All other values, including `True`, non-zero numbers, and non-empty sequences, are implicitly `True`. In the case of the code above, the value `"A"` is a non-empty string, so it is `True`. As `or` requires only one condition to evaluate as `True`, the `print` statement inside of the `if` block is always run.

Note that there are often good reasons why it might be desirable to use implicit truth testing, such as in the check below that runs if some objective is met before time has expired.

```
if objective_met and not time_expired:
    # CODE
```

This is easier to read, and less verbose, than the following:

```
if objective_met == True and time_expired == False:
    # CODE
```

3. REPEATING CODE

Making decisions is only one aspect of a typical computer program. Most programs involve repeating one or more pieces of code, such as performing a calculation or comparison on multiple values. Python provides two main methods for doing this: **counted repetition**, and **conditional repetition**.

3.1. Iterating Over a Sequence

Python can **iterate** (repeat one or more actions) over the elements of any sequence, such as a string or list. The `for` keyword instructs the interpreter that a sequence of values will be assigned to a variable, and that each time this occurs, there will be some code to run. This code is indented, similar to an `if` block. The `in` keyword tells the interpreter to look "in" the a given sequence for these values. For example, the code below counts the number of As in a given string.

```
s = input("Enter something: ")
number_of_As = 0
for char in s:
    if char == "A" or char == "a":
        number_of_As += 1
print("Your string contains", number_of_As, "A's.")
```

The variable `char` takes on the value of each character in `s`. If the input is "Abacus", `char` will be assigned the values "A", "b", "a", "c", "u" and "s" resulting in a count of 2.

3.2. Generating Sequences of Integers Using `range`

To generate a sequence of integers, Python provides the `range` function, which takes up to three arguments: *start*, *stop* and *step*. The range of integers begins at *start*, and ends at the integer immediately before *stop* – that is, *stop* is not included. If two arguments are provided, the two values are *start* and *stop*. If only one is provided, it represents *stop*. In this case, *start* is assigned a value of 0.

```
>>> sequence = list(range(5, 10))
>>> sequence
[5, 6, 7, 8, 9]
>>> sequence = list(range(8))
>>> sequence
[0, 1, 2, 3, 4, 5, 6, 7]
```

If given, the third argument, *step*, acts as a "count by" value. A negative value for *step* counts backward. Note that *start* must be larger than *stop* in this case.

```
>>> sequence = list(range(3, 11, 2))
>>> sequence
[3, 5, 7, 9]
>>> sequence = list(range(10, 4, -1))
>>> sequence
[10, 9, 8, 7, 6, 5]
```

3.3. Counted Repetition

Since `range` generates a sequence of integers, it is useful for repeating code a fixed number of times. Like with any sequence, a `for` loop can be used to repeat a block of code.

```
for i in range(5):
    print("This message is displayed five times.")
```

The variable `i` takes on the integer values 0 through 4 (5 is not included), and so the loop runs five times. Since only one argument is provided to `range`, it acts as the *stop* value. The code can easily be extended to run an arbitrary number of times, provided the argument is an integer value.

```
n = int(input("Run how many times? "))
for i in range(n):
    print("This message is displayed", n, "times.")
```

3.4. Conditional Repetition

Sometimes it is not known beforehand how many times a loop needs to run. In other cases, a loop needs to run as long as one or more conditions are `True`. In these situations, a for loop is not the best option. Instead, use a `while` loop. The code below repeatedly doubles a number until it exceeds one million. As long as the condition `n < 1000000` is `True`, the loop will continue to run.

```
n = int(input("Enter a value of n: "))
while n < 1000000:
    n *= 2
    print(n)
```

Using a `while` loop is an effective method for obtaining valid user input.

```
n = int(input("Enter a positive integer: "))
while n <= 0:
    n = int(input("Invalid value, try again: "))
print("Thank you.")
```

3.5. Flow Control For Loops

Terminating a loop early can be done via the `break` keyword. When a `break` statement is encountered, a loop will immediately stop executing and the program will proceed to the next line of code after the loop. Any code inside of the loop, after the `break` statement, will not be executed.

```python
for i in range(10):
    if i == 4:
        break
    print(i)
```

Unlike a for loop, which will eventually end once the final element in a sequence has been processed, it is possible to create a `while` loop that repeats forever. Such a loop is often called an **infinite loop**. A simple way to do this is to use a condition that is always `True`.

```python
while True:
    s = input("Type something: ")
    print("You typed:", s)
```

Using `break` can cause the loop to terminate.

```python
while True:
    s = input("Type something ('q' to quit): ")
    if s == "q" or s == "Q":
        break
    print("You typed:", s)
```

The `continue` keyword will stop the current iteration of the loop, but will return to the beginning of the loop to begin the next iteration. Like `break`, any code that follows a `continue` statement will not be executed.

```python
for i in range(10):
    if i == 4:
        continue
    print(i)
```

Both `for` and `while` loops can include an optional `else` clause. Any code inside of this clause will execute only if the loop ends when the loop terminates "naturally." In the case of a `for` loop, this means that all values in the sequence have been processed. In the case of a `while` loop, this means that the condition is `False`. The code in an `else` clause will not run if the loop terminates on account of a `break` statement.

```python
import random
for i in range(5):
    random_num = random.randint(1, 10)
    print(random_num)
    if random_num > 8:
        break
else:
    print("No values 8 or greater were generated.")
```

4. SEQUENCES

A good deal of input and output is not based on single values, but on collections of things: characters, numeric values, or other objects. Python has several sequence data types, including strings, tuples and lists, that can be used to manage these collections.

4.1. Strings, Tuples and Lists

A string is a sequence of characters, such as “abc123#$%”. Tuples and lists, on the other hand, are collections of *objects*, including fundamental data types (integers, floating point values, strings) as well as others, such as additional tuples and lists, or even user-created objects. Strings are enclosed in either single or double quites. Tuples can be created by surrounding comma-separated values with round brackets (or by simply separating values with commas), while lists use square brackets.

```
>>> s = "hello"
>>> type(s)
<class 'str'>
>>> T = (1, 2, 3)
>>> type(T)
<class 'tuple'>
>>> L = ["a", "b", "c"]
>>> type(L)
<class 'list'>
```

Many built-in functions operate on sequences. To count the number of characters in a string, or the number of elements in a list or tuple, use `len`.

```
>>> s = "abc123"
>>> len(s)
6
```

Both `max` and `min` operate on sequences. For strings, comparisons are based on the ASCII value of the character, so capital letters are “less than” lowercase ones. For lists and tuples, elements cannot be a combination of numeric and non-numeric values.

```
>>> animal = "Zebra"
>>> min(animal)
'Z'
>>> numbers = [3, 8, 2.1]
>>> max(numbers)
8
>>> chars = ["a", 5, 7.3]
>>> min(chars)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

4.2. Indexing and Slicing

To access a particular character in a string, or a element in a tuple or list, use square brackets with a specific position (or **index**) inside. Python uses **zero-based indexing**; that is, counting starts at zero.

```
>>> s = "abcde"
>>> s[0]
'a'
>>> L = ['cat', 'dog', 'pig']
>>> L[1]
'dog'
```

Attempting to access an non-existant index results in a run-time error.

```
>>> L[99]
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
IndexError: list index out of range
```

Multiple characters or elements can be obtained by specifying a **slice** for a string, tuple or list. Slicing notation is similar to that of `range` but uses colons to separate the *start*, *stop* and *step* values. Both *start* and *stop* can be omitted to start at the beginning of the sequence, or finish at the end.

```
>>> s[1:3]
'bc'
>>> s[:4]
'abcd'
>>> s[::2]
'ace'
```

A sequence can be reversed using a negative *step* value.

```
>>> s = "Python"
>>> print(s[::-1])
nohtyP
```

4.3. Mutable and Immutable Sequences

A data type that can be modified after it has been created is **mutable**. One that cannot be modified is **immutable**. Most of the built-in data types are immutable: integers, Booleans, floating point values, strings and tuples. Their values can only be changed by overwriting them.

```
>>> s = "hello"
>>> s[0] = "c"
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'str' object does not support item assignment"
```

Lists, sets and dictionaries, on the other hand, are mutable. It is possible to change individual characters or elements.

```
>>> L = [7, 1, 3]
>>> L[0] = 9
>>> L
[9, 1, 3]
>>> L[-1] = 5
>>> L
[9, 1, 5]
```

4.4. String Methods

A function that is associated with a particular data type is called a **method**. There are several methods available to use with strings that can provide information about the string or modify it.

When a string contains an instance of a smaller string, that smaller string is called a **substring**. To check whether or not a substring exists in a given string, use the `in` keyword.

```
>>> s = "programming"
>>> "gram" in s
True
>>> "xyz" in s
False
```

To locate the left-most index of a substring, there are several options available for strings. `find` and `rfind` will search the string from the left and right-hand sides respectively. `index` and `rindex` do the same, but they will cause a run-time error if a substring is not found, whereas `find` and `rfind` will return a value of -1 instead. For this reason, it's advisable to verify that the substring exists by using `in` before attempting to use index on a string.

```
>>> s = "abcabcabc"
>>> s.find("bc")
1
>>> s.rfind("bc")
7
>>> s.find("z")
-1
>>> s.index("bc")
1
>>> s.index("z")
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: substring not found
```

To count the number of non-overlapping instances of a substring, use `count`.

```
>>> s = "aaaabbbcc"
>>> s.count("b")
3
>>> s.count("aa")
2
```

Python also provides several methods for modifying the characters of a string. `upper` and `lower` will convert any alphabetic characters to the specified case. `replace` will take a target substring and substitute them with characters in a new substring.

```
>>> s = "123 Streep Street"
>>> s.upper()
'123 STREEP STREET'
>>> s.replace("e", "o")
'123 Stroop Stroot'
```

The `strip`, `lstrip` and `rstrip` methods remove characters from one or more ends of a string. They take as arguments a single string of characters, and will remove all such characters until the first character non specified is encountered.

```
>>> s = "*aabc*aba**"
>>> s.strip("*a")
'bc*ab'
>>> s.rstrip("*a")
'*aabc*ab'
```

A full list of string methods can be found in the official documentation.

4.5. Splitting and Joining Strings

Strings can be broken apart on certain characters using the `split` method. All pieces of the string are stored in a list. If no string is provided as an argument, `split` will break up a string on whitespace (spaces, tabs, and so on).

```
>>> s = "This is a string."
>>> s.split()
['This', 'is', 'a', 'string.']
```

A sequence (list, tuple, set) of strings can be joined together using the `join` method. Since it is a string method, it operates on the string itself and not the sequence.

```
>>> words = ["apple", "banana", "cherry"]
>>> "**".join(words)
'apple**banana**cherry'
```

It is important to ensure that the sequence contains only strings. Other data types, including numeric ones, will cause an error if `join` is used on them.

4.6. List Methods

Unlike tuples, lists are mutable, and so it is possible to add and remove elements from them. To add an element to the tail end of a list, use `append`. To add an element in a specific index, use `insert`. These two methods are referred to as **adding by value** and **adding by index** respectively.

```
>>> L=['a', 'b', 'c']
>>> L.append('d')
>>> L
['a', 'b', 'c', 'd']
>>> L.insert(0, 'e')
>>> L
['e', 'a', 'b', 'c', 'd']
```

Removal can also be done by value or by index. To remove an existing element in a list, use `remove`. Use `pop` to remove an element from a specified index and, optionally, assign it to a variable. The `del` function that applies to all objects will also remove an element by index. Note that the given element must exist, or a run-time error will occur.

```
>>> L.remove('a')
>>> L
['e', 'b', 'c', 'd']
>>> L.pop(1)
>>> L
['e', 'c', 'd']
>>> del(L[-1])
>>> L
['e', 'c']
```

Like with strings, the simplest way to check if an element is in a list is to use `in`. To obtain the index of a specific element, use `index`. There are no `find`, `rfind` or `rindex` methods like there are with strings. Calling `index` with a non-existant element will cause an error. To count the number of occurrences of an element, use `count`.

```
>>> 'A' in L
False
>>> L.index('c')
2
>>> L.count('b')
1
```

As always, all of the methods for lists are described in the official documentation. See the sections "common methods" and "mutable methods" for the entire suite of options.

4.7. List Comprehensions

Lists can be populated using loops, like in the example below.

```
>>> L = []
>>> for i in range(5):
    L.append(i)

>>> L
[0, 1, 2, 3, 4]
```

An alternative method is to use a **list comprehension**, which uses a loop-like notation to populate the list. The loop in the previous example can be replaced by the code below.

```
>>> L = [x for x in range(5)]
>>> L
[0, 1, 2, 3, 4]
```

In general, a simple list comprehension has the format `[`*`operation`*` for `*`element`*` in `*`sequence`*`]`. In the example above, the *sequence* was the `range` object containing the integers 1 through 5, which assigned the values to the *element x*. The *operation* in this case was to do nothing and simply use the value of *x* itself. Below is an example that doubles all even values in a list.

```
>>> integers = [7, 4, 8, 1, 12]
>>> double_evens = [2*n for n in integers if n%2 == 0]
>>> double_evens
[8, 16, 24]
```

Note that in the example above, conditions can be attached to the loop. For more information and examples, see the official documentation.

5. FUNCTIONS AND MODULARITY

A **function** is a piece of code that can be referenced and run within a larger program. Three of the main advantages to breaking up a larger program into functions is organization, code reuse, and maintenance. This allows for the same, or similar, actions to be run multiple times without the need to repeat code in different locations.

5.1. Creating Functions

The `def` keyword tells the interpreter that the code that follows is a function. Each function must have a name that follows standard variable-naming conventions, and a sequence of **arguments** (possibly none). The code below counts the number of As in a given string, *s*.

```
def count_capitals(s):
    count = 0
    for char in s:
        if char >= "A" and char <= "Z":
            count += 1
    print(count)
```

To call a function from the main program, use its name while passing it any required arguments.

```
user_input = input("Enter a string of characters: ")
count_capitals(user_input)
```

Note that the names of arguments do not need to match. Python will associate any values with their new names upon receiving them.

5.2. Variable Scope

Variables created in the main program are *visible* to functions, even if not passed as arguments. The code below will display twice the value of *n*, even though n is not created in the function.

```
def double():
    print(n*2)

n = int(input("Enter an integer: "))
double()
```

While the value of *n* can be seen inside of the function, it cannot be modified by it. The following code attempts to double the value of *n* using the multiplicative assignment operator.

```
def double_num():
    n *= 2
    print(n)

n = int(input("Enter an integer: "))
double_num()
```

This results in an error.

```
Traceback (most recent call last):
  File "/home/jgarvin/program.py", line 5, in <module>
    double_num()
  File "/home/jgarvin/program.py", line 2, in double_num
    n *= 2
UnboundLocalError: local variable 'n' referenced before assignment
```

In order to modify the value of *n*, it must be passed to the function as an argument. Alternatively, a local variable (possibly with the same name) can be explicitly created in the function, and its value modified.

Variables created inside of a function cannot be seen by the main program. A similar error will occur if an attempt is made. This is known as **variable scope**.

5.3. Returning Values

If the main program needs access to a value that was modified in a function, then the function must send the updated value back. The `return` keyword does just this. It is possible to return multiple values if they are separated by commas (they will be packed into a tuple).

```
def sum_and_product(a, b):
    s = a + b
    p = a * b
    return s, p

a = int(input("Enter an integer: "))
b = int(input("Enter another: "))
s, p = sum_and_product(a, b)
print("The sum is", s, "and the product is", p)
```

In most cases, returning values is preferrable to using global variables (using the `global` keyword).

5.4. Default Values and Argument Packing/Unpacking

Arguments can be assigned default values. The following function draws a line of dashes, with its length specified by the user.

```
def draw_line(length=10):
    print("-"*length)

draw_line()
draw_line(20)
```

The first call to `draw_line` does not pass any arguments to the function, so it will display ten dashes. The second call overrides the default value, and results in twenty dashes.

If a default value is specified for any argument, all subsequent arguments must also have default values associated with them.

```python
def print_name(num_times=5, name="Jon"):
    for i in range(num_times):
        print(name)

print_name()
print_name(10)
print_name("Bill")
```

The third call to `print_name` causes a run-time error because the `for` loop expects an integer value as the first argument. It is possible to use keyword arguments to specify only the name, such as `print_name(name="Bill")`. This is equivalent to `print_name(5, "Bill")`.

Some functions require multiple arguments to operate correctly. For example, consider the function below that calculates the net profit given the gross revenue, expenses, and tax rate.

```python
def calculate_profit(revenue, expenses, tax):
    profit = (revenue-expenses) * (1-tax)
    return profit
```

Now assume that the values for `revenue`, `expenses` and `tax` are already stored in list or tuple (perhaps they were read from a spreadsheet or other text file), as in the list `values` which contains `[revenue, expenses, tax]`. One way to call the function would be to pass the individual elements as separate arguments.

```python
profit = calculate_profit(values[0], values[1], values[2])
```

An alternative method is to use **argument unpacking**, whereby Python will “unpack” a sequence of values as separate arguments. In the function call, prefix the variable with a * to indicate that it should be unpacked. Note that the number of elements must be equal to the number of arguments, or an error will occur.

```python
profit = calculate_profit(*values)
```

Similarly, a function can “pack” multiple values into a single sequence. The code below takes multiple integer values, then determines their sum using the `sum` function for sequences.

```python
def find_sum(*nums):
    total = sum(nums)
    return total

s = find_sum(1, 2, 3, 4, 5)
print(s)
```

5.5. Modules

If functions are to be reused across multiple projects, it is good practice to bundle together them as a module. This avoids code duplication, and is generally easier to maintain as there is only one source to modify. A module is just a regular Python file containing variable declarations and functions. For example, the following code is saved in the file `dice.py`.

```
import random
def roll(sides=6, num_dice=1):
    dice = []
    for r in range(num_dice):
        dice.append(random.randint(1, sides))
    return dice
```

Accessing the contents of `dice.py` is simply a matter of importing it into the main program, and prefixing any function calls with the module name. The code below is saved in a different file, `dice_roller.py`.

```
import dice
rolls = dice.roll(10, 3)
print("You rolled:", ",".join(rolls))
```

Note that the call to `roll` is prefixed by the name of the module, `dice`. Omitting this will result in a run-time error. It is possible to assign an alias to a module so that it can be referred to using a shorter prefix.

```
>>> import random as r
>>> r.randint(1, 6)
4
```

Doing this may obscure the source of the functions for the reader, so use with caution.

6. FORMATTING OUTPUT

In addition to calculating values and performing other simple actions, many programs require output to conform to certain criteria. These include displaying data in a tabular form, rounding values to a certain number of decimal places, and so on. In other situations, the case of text must be altered to fit a certain format. There are many ways to do this in Python.

6.1. Methods to Transform Case

Some of the more common methods to transform the case of strings are `upper` and `lower`, which convert all alphabetic characters to the specified case. Other methods include `capitalize`, `title`, `swapcase`, among others.

```
>>> s = "Jon Garvin"
>>> s.upper()
'JON GARVIN'
>>> s.swapcase()
'jON gARVIN'
```

Both `upper` and `lower` are useful for reducing the number of comparisons in an `if` block or a loop. Consider the example below.

```
option = input("Enter your choice, option A or B: ")
if option == "a" or option == "A":
    print("You chose option A.")
if option == "b" or option == "B":
    print("You chose option B.")
else:
    print("That's not a valid option.")
```

This can be replaced by the following code instead.

```
option = input("Enter your choice, option A or B: ").lower()
if option == "a":
    print("You chose option A.")
if option == "b":
    print("You chose option B.")
else:
    print("That's not a valid option.")
```

As usual, see the documentation for a full list of available methods.

6.2. Basic String Formatting Methods

Sometimes it is desirable to align text in a certain way (left- or right-justified, or centred). Often, this can be accomplished using a mix of spaces and tabs (`\t`), but Python also provides methods for aligning output via the `ljust`, `rjust` and `center` methods. These each take up to two arguments: one specifying the width of the displayed string, including any whitespace or fill characters, and the other specifying a fill character.

```
>>> s = "Python"
>>> s.rjust(20)
'              Python'
>>> s.center(20, '*')
'*******Python*******'
```

6.3. Formatting Using the `format` Method

A more general-purpose formatting system is provided via the `format` method. Strings that use format use curly braces where arguments will be inserted into a string.

```
>>> num = 42
>>> "The answer is {}.".format(num)
'The answer is 42.'
```

Inside of the curly braces are symbols and values for many options, including the width of the displayed text and a specifier for its data type (`d` for integers, `f` for floating points, and so on). Left-, right- and centre alignmnent is handled by the `<`, `>` and `^` symbols respectively. Floating point values can be rounded by specifying the number of decimal places.

```
>>> x = 7
>>> y = 5.2981
>>> "{:>5d}".format(x)
'    7'
>>> "{:^10.2f}".format(y)
'   5.30   '
```

The `format` mini-language specification page provide several other examples.

6.4. f-Strings

Introduced in Python 3.6, formatted string literals (or **f-strings** for short) provide some of the powerful options that the format method in a more concise manner. f-strings are indicated by prefixing a string with `f`, then using curly braces containing values and options.

```
>>> animal = "cat"
>>> f"I have a {animal}."
'I have a cat.'
```

Since they are evaluated at run-time, f-strings can use any Python expression inside of the curly braces, including mathematical operations or function calls.

```
>>> a = 3
>>> b = 7
>>> f"{a} * {b} = {a*b}"
'3 * 7 = 21'
```

Like `format`, f-strings can handle alignment using `<`, `>` and `^`.

```
>>> colours = ['red', 'green', 'blue']
>>> for colour in colours:
    print(f"{colour:>10}")

       red
     green
      blue
```

Rounding is also similar.

```
>>> pi = 3.14159
>>> f"{pi:.2f}"
'3.14'
```

f-strings are faster than `format`, and are the preferred method of formatted output in Python. To read up on how to use f-strings, see the documentation.