ICS3U: String Basics

Recall that a string is a sequence of characters. These characters may be letters, numbers, whitespace, or special characters. We have already seen how we can concatenate two strings using the `+` operator, or repeat a string several times using `*`.

```
>>> print("tro"+"lo"*3)
trololo lo
```

Often, we use concatenation to "build up" a string, beginning with an empty string. Try running the code below to see how this can work.

```
s = ""
while True:
   c = input("Enter a letter (or press ENTER to quit): ")
   if c == "":
      break
   else:
      s += c
print(s)
```

Sometimes we are interested in how many characters make up a given string. In Python, we can use the `len` built-in function to determine how long a string is. This includes all characters: letters, numbers, whitespace and special characters.

```
>>> s = "The answer is 42."
>>> len(s)
17
```

Each character in a string is in a certain position, known as an **index** (plural **indices**). For instance, consider the string 'CODE'. The index of the letter 'C' is 0, since it is the first character in the string. It is *not* 1, as some might expect: Python uses **zero-based indexing** for *all* sequence types, including strings, tuples, lists, rangesm and so on. The letters 'O', 'D' and 'E' have indices 1, 2 and 3 respectively. Note that the index of the final character is *always* one less than the length of a string.

To access a specific character in a string, we can use a process called **indexing** wherein we specify the desired index. Python uses square brackets for this: `[i]` denotes an index of i. We can apply this to a given string to obtain a single character.

```
>>> s = "Hello"
>>> s[0]
'H'
>>> s[1]
'e'
```

Trying to access a character using an index does not exist will produce a run-time error instead.

```
>>> s[5]
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    s[5]
IndexError: string index out of range
```

Python also lets us access individual characters in a string in a “backwards” manner by using negative indices. The last character has an index of -1, the second last an index of -2, and so on. This makes it easy to quickly obtain the final character in a string, regardless of its length.

```
>>> sentence = "This is a very, very, very long string full of many characters!"
>>> sentence[-1]
'!'
```

Since both strings and `range` objects use zero-based indexing, it is possible to iterate over a string using a `for` loop:

```
word = "Python"
for x in range(len(word)):
   print(word[x])
```

Of course, we have already seen how we can do this using `in`.

```
word = "Python"
for letter in word:
   print(letter)
```

There are advantages to both methods. An interesting application of indexing is parsing the individual digits of a number. It is possible to obtain each digit with some modular arithmetic (using `%` and `//`). For example, if we repeatedly use both modulus and division by 10, we can identify the digits of a number in reverse.

```
num = int(input("Enter an integer: "))
while num > 0:
   print(num % 10)
   num //= 10
```

By representing a number as a string, however, each digit will have its own index. The code below separates the digits, *in order*, using indexing. Note that the `input` statement is not nested in a call to `int` as it was in the previous code. It also preserves the value of `num`.

```
num = input("Enter an integer: ")
print("The digits in the number are ", end="")
for i in range(len(num)-1):
   print(num[i]+", ", end="")
print("and", num[-1])
```

We can also reverse a string using a simple indexed loop.

```
s = input("Enter some text: ")
new = ""
for i in range(len(s)-1, -1, -1):
   new += s[i]
print(s, "reversed is", new)
```

This isn’t the most Pythonic way to do this, as we will soon see, but it works.

Data can be stored using a variety of **encodings**. One of the original character encoding schemes was ASCII (American Standard Code for Information Interchange), which uses the values 0-127 to represent letters in the Roman alphabet (both upper- and lowercase), the digits 0-9, as well as some special characters including punctuation, spaces, and so on. While ASCII has been replaced by Unicode these days, to provide a greater number of characters, it is still widely-used in many programming languages.

Python uses `ord` to obtain the decimal value of an ASCII-encoded character, and `chr` to obtain the character given a decimal value. Below is an example of each.

```
>>> ord("A")
65
>>> chr(67)
'C'
```

Note that it is possible to use values greater than 127 when using `chr`. This is because both `ord` and `chr` will also work with Unicode characters. Try some larger values and see what is displayed.

Unicode characters can also be displayed in Python by using a `\u` escape code (for 16-bit hexadecimal values) or `\U` (for 32-bit hexadecimal values), followed by the hexadecimal value of the unicode character to be displayed. For example, to output the capital Greek letter Omega (Ω) which has the hexadecimal value 03A9, type the following.

```
>>> print('\u03A9')
Ω
```

This can be useful if you need to display some text in another language, or if you wish to take advantage of the symbols available in Unicode to simulate game pieces, draw arrows, create boxes, etc. Be aware that there may be spacing issues associated with Unicode characters.