> *Useful stuff, string, no trouble there*

Data Structures

Sequences

A sequence can be thought of as an ordered collection of data – i.e. it has a 0^{th} element, a first element, a second element and so on.

Note that Python, and many other programming languages, start counting from 0 rather than 1.

A typical operation on any kind of a sequence is to be able to access the i^{th} element. Another operation is to determine how many elements there are in the sequence. This also leads to the ability to walk through the sequence from start to end. This is known as **iterating** through the sequence. All sequence-based objects that have this capability are known as **iterables**.

A string can be considered as a specialisation of a sequence as it represents a series of ordered characters. We will use strings as our first example of using ADTs.

Strings

As we have seen before a string can be made using the quotation marks. Python interprets anything held within a pair of "" to be a string. This can be thought of as the **constructor** of a string.

Because strings are specialisations of sequences, then we expect to be able to get the i^{th} character of a string and find out how long a string is. To be able to access the characters in a string it is possible to do what is called **indexing**. This is done using `[ ]` notation – this is the **accessor**. To be able to find the length of a string we have the len function.

Note that there are other operations we want to perform on strings that may not make sense for other specialisations of sequences.

Here are some examples of indexing and finding the of length strings in action:

```
>>> s = "spam"
>>> len(s)
4
>>> s[0]
's'
>>> s[3]
'm'
>>> s[4]
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    s[4]
IndexError: string index out of range
>>> s[-1]
'm'
>>> s[-2]
'a'
```

```
>>> i = 0
>>> while i < len(s) :
        print(s[i])
        i += 1

s
p
a
m
```

Note the 'string index out of range' error in the fourth example – **there is no fourth element as Python starts counting at 0**. Also notice that negative numbers can be used in indexing. The use of negative numbers starts the indexing at the **end** of the string. So -1 refers to the last element in the string, -2 refers to the second last, and so on. In the last example we use a while loop to iterate over and print every character in the string. Remember **i += 1** is the same as **i = i + 1**.

In Python, there is no separate 'character type'. Characters in Python are represented as strings of length 1. We will use the term 'character' to mean a string of length 1.

Strings are **immutable sequences** – i.e. it is not possible to change the data in a string. We will cover **mutable** and **immutable** objects and sequences in the following weeks' notes.

len Syntax

```
len(sequence)
```

Semantics

Returns the length of (number of objects in) the sequence.

For

In the example above, a while loop was used to iterate over the characters of a string. Iterating over the elements of a sequence is such a common thing to do that Python has a special construct for this purpose – the **for loop**.

Instead of using the while loop as we did earlier we could use a for loop instead, as a string is an iterable object.

```
>>> s = "spam"
>>> for i in s :
        print(i)

s
p
a
m
```

Syntax

```
for var in sequence :
    body
```

where *var* is a variable and *sequence* is a sequence.

Semantics

Iterate over *sequence* assigning *var* in turn to each element of the sequence. For each element, execute *body*. The body typically contains occurrences of *var*. The body must follow the indentation rules of Python that we have seen before.

As a simple example of the use of a for loop we now write a function to determine if a given character is in a string. Here is the definition:

```
def is_in(char, string) :
    """Return True iff 'char' is in 'string'.

    Parameters:
        char (string): The character being searched for.
        string (string): The string being searched.

    Return:
        bool: True if 'char' is in 'string'. False otherwise.
    """
    for c in string :
        if c == char :
            return True
    return False
```

Here is an example of `is_in`:

```
>>> spam = "spam"
>>> is_in("s", spam)
True
>>> is_in("d", spam)
False
```

The for loop iterates over the characters in the string, comparing each in turn with the supplied character. If a match is found, `True` is returned. If we iterate through all the elements without finding a match, then the for loop terminates and the following command is executed – returning `False`.

Determining if a particular item is an element of a sequence is a common operation, and so Python has a built-in operator **in** that does the job for us – so we don't need the previous function. Actually, for strings, **in** does more than the previous function as we can see in the following examples.

```
>>> 's' in 'spam'
True
>>> 'x' in 'spam'
False
>>> 'sp' in 'spam'
True
>>> 'sa' in 'spam'
False
>>> 'pa' in 'spam'
True
```

Note how the `in` keyword is used in for loops as well as boolean tests. It's important to be aware of both uses of `in` and how to use it in both cases.

Slice It Up

Earlier we saw how we can access the i^{th} element using square brackets. We can do more! We can also use the square brackets to do **slicing** – i.e. extracting subsequences. Here are some examples.

```
>>> s = 'spam'
>>> s[1:2]
'p'
>>> s[1:3]
'pa'
>>> s[:3]
'spa'
>>> s[1:]
'pam'
>>> s[:-2]
'sp'
>>> s[-3:]
'pam'
>>> s[:]
'spam'
```

The idea is to supply two indices separated by a colon. So when we write `s[n:m]`, we mean the substring from the n^{th} index **up to, but not including**, the m^{th} index. If the first index is missing we get the slice starting at the beginning of the string or sequence. If the second index is missing we get the slice ending at the last element. In the last case we actually generate a **copy** of the original string.

We can also write `s[n:m:k]`. This means the substring from n to m in steps of k. Here are some examples.

```
>>> sp = 'Lovely Spam'
>>> sp[1:10:2]
'oeySa'
>>> sp[0:8:3]
'Le '
>>> sp[-10:2:2]
'o'
>>> sp[-10:10:3]
'olS'
>>> sp[:7:4]
'Ll'
>>> sp[2::3]
'vyp'
>>> sp[::3]
'Le a'
>>> sp[10:1:-2]
'mp lv'
>>> sp[-3:2:-3]
'py'
>>> sp[::-1]
'mapS ylevoL'
```

The first four examples show starting from one index and ending at another using a step size. The next three examples show ways of starting at the beginning or ending at the end or both using a step size. The last three are examples of using negative step sizes.

Slicing Syntax

*sequence***[n:m:k]**

Semantics

Returns the elements in the sequence from `n` up to but not including `m` in steps of size `k`. If `k` is not included then step size defaults to 1. To get a backwards segment of the sequence then `m` < `n` and `k` must be negative.

To finish off this section we look at two programming problems involving lists. The first problem is, given a character and a string, find the index of the first occurrence of the character in the string.

```
def find(char, string) :
    """Return the first i such that string[i] == 'char'

    Parameters:
        char (string): The character being searched for.
        string (string): The string being searched.

    Return:
        int: Index position of 'char' in 'string',
             or -1 if 'char' does not occur in 'string'.
    """
    i = 0
    length = len(string)
    while i < length :
        if char == string[i] :
            return i
        i += 1
    return -1
```

We use a while loop to iterate over the elements of the string. Either we find an occurrence of the character — in which case we immediately return that index, or we get to the end of the string without finding the character and return -1.

Here are some tests of `find`.

```
>>> s = "spam"
>>> find('m', s)
3
>>> find('s', s)
0
>>> find('x', s)
-1
```

Tuples

A tuple is a comma separated list of values. Tuples are useful for storing data that is required to not be modified. As tuples are immutable, **they cannot be modified**, they are useful for this type of operation.

Here are some examples using tuples.

```
>>> type((2,3,4))
<class 'tuple'>
>>> point = (2,3)
>>> point
(2, 3)
>>> x, y = point
>>> print(x, y)
2 3
>>> x, y = y, x
>>> print(x, y)
3 2
```

The second example assigns a tuple to the variable `point`. This tuple represents an x, y coordinate, one of the more common uses of tuples. (Formally, we say that the name `point` refers to the tuple (2,3) in memory.) The third is an example of **tuple unpacking**, also called **parallel assignment** — it assigns `x` and `y` respectively the values of the first and second components of the tuple. The last example uses parallel assignment to swap the values of two variables.

Tuples can also be indexed and sliced in the same way as strings. Here is an example. The tuple in this example represents a person's details. Using a tuple is an easy way of storing multiple values together to represent the information of a person.

```
>>> john = ("John", "Cleese", "Ministry of Silly Walks", 5555421, "27/10")
>>> john[2]
'Ministry of Silly Walks'
>>> john[4]
'27/10'
>>> john[:2]
('John', 'Cleese')
```

Multiple Outputs

Tuples are also useful for using multiple values, for example in for loops and return statements. It is possible to rewrite the `find` function using a for loop and the `enumerate` function. Calling `enumerate` on a sequence will create a sequence of tuples containing a counter and the values in the sequence. By default, the counter starts at 0, so it can be used to generate tuples that contain a value of the sequence and the index of that value. The following is a couple of examples using `enumerate`.

```
>>> s = "I like Spam"
>>> for i, c in enumerate(s) :
    print(i, c)

0 I
1
2 l
3 i
4 k
5 e
6
7 S
8 p
9 a
10 m
>>>
>>>
>>> for i, c in enumerate(s, 3) :
    print('number:', i, 'character:', c)

number: 3 character: I
number: 4 character:
number: 5 character: l
number: 6 character: i
number: 7 character: k
number: 8 character: e
number: 9 character:
number: 10 character: S
number: 11 character: p
number: 12 character: a
number: 13 character: m
```

Notice that the for loop uses tuple unpacking, by assigning the items of the tuple to the variables `i` and `c`. The first example simply prints out the index and character as a tuple pair. The second gives a second argument to `enumerate`, which is the starting value for the counter.

`enumerate` Syntax

Either of these forms can be used:

```
enumerate(sequence)
enumerate(sequence, start)
```

Semantics

Generates a sequence of `(count, value)` tuples, with an increasing count and the values of the sequence. The count starts at `start`, or at 0 if `start` is not given. More precisely, the following sequence is generated:

```
enumerate(seq) => (0, seq[0]), (1, seq[1]), (2, seq[2]), ...
enumerate(seq, start) => (start, seq[0]), (start+1, seq[1]),
                         (start+2, seq[2]), ...
```

> If the counter starts from 0, then the count is the same as the index of the value in the sequence. This is the most common use of `enumerate`.

Now we can rewrite `find` as below:

```
def find(char, string) :
    """Return the first i such that string[i] == char

    Parameters:
        char (string): The character being searched for.
        string (string): The string being searched.

    Return:
        int: Index position of 'char' in 'string',
             or -1 if 'char' does not occur in 'string'.
    """
    for i, c in enumerate(string) :
        if c == char :
            return i
    return -1
```

This function now goes through each character in the string using a for loop with enumerate. The body of the for loop is to check if the current character, `c`, is the same as `char`. If it is then the current index `i` is returned. Otherwise the for loop moves onto the next character in the string. If the for loop ends then -1 is returned.

Performing test cases on find.py we can see that it has the same functionality as before.

```
>>> s = "spam"
>>> find('m', s)
3
>>> find('s', s)
0
>>> find('x', s)
-1
```

Returning a Tuple

Now we are going to write our own function that returns multiple outputs. Our function is going to take a character and a string and split the string in two, at the first occurrence of the character. In this case we want to write a function that will return three strings - the string up to (but not including) the character; the character; and the string after the character.

Our programming problem can be solved by using `find` in combination with slicing.

```
def partition(char, string) :
    """Return 'string' split at 'char'.

    The returned result is a tuple consisting of three strings that
    partition 'string' at 'char' - i.e. the substring before the first occurrence
    of 'char', 'char', and the substring after the first occurrence of 'char'.
    If 'char' does not occur in 'string' then the first component returned
    is the entire 'string' and the last two components are empty strings.

    Parameters:
        char (string): The character used to partition string.
        string (string): The string being partitioned.

    Return:
        tuple<tr, str, str>: sub-string before char, char, sub-string after char;
                             or 'string', "", "".
    """
    index = find(char, string)
    if index == -1 :
        return string, '', ''
    else :
        return string[:index], char, string[index+1:]
```

Here are some tests of `partition`.

```
>>> spam = 'spam'
>>> partition('s', spam)
('', 's', 'pam')
>>> partition('p', spam)
('s', 'p', 'am')
>>> partition('m', spam)
('spa', 'm', '')
>>> partition('x', spam)
('spam', '', '')
```

In next week's notes we will see that the `find` and `partition` functions, like the `is_in` function are already part of Python as part of the string ADT interface.