

## Chapter 4

# Strings

### 4.1 A string is a sequence

A string is a **sequence** of characters. You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

The second statement selects character number 1 from `fruit` and assigns it to `letter`.

The expression in brackets is called an **index**. The index *indicates* which character in the sequence you want (hence the name).

But you might not get what you expect:

```
>>> print letter
a
```

For most people, the first letter of 'banana' is b, not a. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> print letter
b
```

So b is the 0th letter (“zero-eth”) of 'banana', a is the 1th letter (“one-eth”), and n is the 2th (“two-eth”) letter.

You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.0]
TypeError: string indices must be integers
```

## 4.2 len

len is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the `IndexError` is that there is no letter in 'banana' with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:

```
>>> last = fruit[length-1]
>>> print last
a
```

Alternatively, you can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

## 4.3 String operations

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers, so the following are illegal:

```
'2'-'1'      'eggs'/'easy'      'third'*'a charm'
```

The `+` operator does work with strings, but it might not do exactly what you expect: it performs **concatenation**, which means joining the strings by linking them end-to-end. For example:

```
first = 'throat'
second = 'warbler'
print first + second
```

The output of this program is throatwarbler.

The `*` operator also works on strings; it performs repetition. For example, `'Spam'*3` is `'SpamSpamSpam'`. If one of the operands is a string, the other has to be an integer.

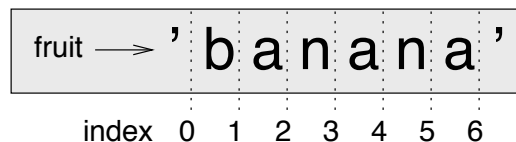
On one hand, this use of `+` and `*` makes sense by analogy with addition and multiplication. Just as  $4*3$  is equivalent to  $4+4+4$ , we expect `'Spam'*3` to be the same as `'Spam'+ 'Spam'+ 'Spam'`, and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition and multiplication have that string concatenation and repetition do not?

## 4.4 String slices

A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>> print s[0:5]
Monty
>>> print s[6:13]
Python
```

The operator `[n:m]` returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last. This behavior is counter-intuitive, but might help to imagine the indices pointing *between* the characters, as in the following diagram:



If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string. Thus:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

**Exercise 4.1.** *Given that `fruit` is a string, what does `fruit[:]` mean?*

## 4.5 Strings are immutable

It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

The “object” in this case is the string and the “item” is the character you tried to assign. For now, an **object** is the same thing as a value, but we will refine that definition later. An **item** is one of the values in a sequence.

The reason for the error is that strings are **immutable**, which means you can’t change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print new_greeting
Jello, world!
```

This example concatenates a new first letter onto a slice of `greeting`. It has no effect on the original string.

## 4.6 string methods

A **method** is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the method `upper` takes a string and returns a new string with all uppercase letters:

Instead of the function syntax `upper(word)`, it uses the method syntax `word.upper()`.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

This form of dot notation specifies the name of the method, `upper`, and the name of the string to apply the method to, `word`. The parentheses indicate that this method has no parameters.

A method call is called an **invocation**; in this case, we would say that we are invoking `upper` on the `word`.

The string method named `find` is the opposite of the `[]` operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns `-1`.

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index
1
```

In this example, we invoke `find` on `word` and pass the letter we are looking for as a parameter.

The `find` method can find substrings, not just characters:

```
>>> word.find('na')
2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

And as a third argument where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because `b` does not appear in the index range from 1 to 2 (not including 2).

**Exercise 4.2.** *Another useful string method is called `count`. Read the documentation of this method and write an invocation that counts the number of `as` in `'banana'`. Hint: there are three.*

## 4.7 Glossary

**object:** Something a variable can refer to. For now, you can use “object” and “value” interchangeably.

**sequence:** An ordered set; that is, a set of values where each value is identified by an integer index.

**item:** One of the values in a sequence.

**index:** An integer value used to select an item in a sequence, such as a character in a string.

**slice:** A part of a string specified by a range of indices.

**empty string:** A string with no characters and length 0, represented by two quotation marks.

**concatenate:** To join two operands end-to-end.

**immutable:** The property of a sequence whose items cannot be assigned.

**traverse:** To iterate through the items in a sequence, performing a similar operation on each.

**search:** A pattern of traversal that stops when it finds what it is looking for.

**counter:** A variable used to count something, usually initialized to zero and then incremented.

**method:** A function that is associated with an object and called using dot notation.

**invocation:** A statement that calls a method.

## 4.8 Exercises