

5.1 Strings

5.1.1 A compound data type

So far we have seen built-in types like `int`, `float`, `bool`, `str` and we've seen lists and pairs. Strings, lists, and pairs are qualitatively different from the others because they are made up of smaller pieces. In the case of strings, they're made up of smaller strings each containing one **character**.

Types that comprise smaller pieces are called **compound data types**. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts. This ambiguity is useful.

5.1.2 Working with strings as single things

We previously saw that each turtle instance has its own attributes and a number of methods that can be applied to the instance. For example, we could set the turtle's color, and we wrote `tess.turn(90)`.

Just like a turtle, a string is also an object. So each string instance has its own attributes and methods.

For example:

```
>>> our_string = "Hello, World!"
>>> all_caps = our_string.upper()
>>> all_caps
'HELLO, WORLD!'
```

`upper` is a method that can be invoked on any string object to create a new string, in which all the characters are in uppercase. (The original string `our_string` remains unchanged.)

There are also methods such as `lower`, `capitalize`, and `swapcase` that do other interesting stuff.

To learn what methods are available, you can consult the Help documentation, look for string methods, and read the documentation. Or, if you're a bit lazier, simply type the following into an editor like Spyder or PyScripter script:

```
1 our_string = "Hello, World!"
2 new_string = our_string.
```

5.1.3 Working with the parts of a string

The **indexing operator** (Python uses square brackets to enclose the index) selects a single character substring from a string:

```
>>> fruit = "banana"
>>> letter = fruit[1]
>>> print(letter)
```

The expression `fruit[1]` selects character number 1 from `fruit`, and creates a new string containing just this one character. The variable `letter` refers to the result. When we display `letter`, we could get a surprise:

```
a
```

Computer scientists always start counting from zero! The letter at subscript position zero of "banana" is b. So at position [1] we have the letter a.

If we want to access the zero-eth letter of a string, we just place 0, or any expression that evaluates to 0, inbetween the brackets:

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

The expression in brackets is called an **index**. An index specifies a member of an ordered collection, in this case the collection of characters in the string. The index *indicates* which one you want, hence the name. It can be any integer expression.

We can use `enumerate` to visualize the indices:

```
>>> fruit = "banana"
>>> list(enumerate(fruit))
[(0, 'b'), (1, 'a'), (2, 'n'), (3, 'a'), (4, 'n'), (5, 'a')]
```

Do not worry about `enumerate` at this point, we will see more of it in the chapter on lists.

Note that indexing returns a *string* — Python has no special type for a single character. It is just a string of length 1.

We've also seen lists previously. The same indexing notation works to extract elements from a list:

```
>>> prime_numbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
>>> prime_numbers[4]
11
>>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
>>> friends[3]
'Angelina'
```

5.1.4 Length

The `len` function, when applied to a string, returns the number of characters in a string:

```
>>> word = "banana"
>>> len(word)
6
```

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

```
1 size = len(word)
2 last = word[size]      # ERROR!
```

That won't work. It causes the runtime error `IndexError: string index out of range`. The reason is that there is no character at index position 6 in "banana". Because we start counting at zero, the six indexes are numbered 0 to 5. To get the last character, we have to subtract 1 from the length of word:

```
1 size = len(word)
2 last = word[size-1]
```

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

As you might have guessed, indexing with a negative index also works like this for lists.

5.1.5 Traversal and the `for` loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way (a very bad way) to encode a traversal is with a `while` statement:

```
1 ix = 0
2 while ix < len(fruit):
3     letter = fruit[ix]
4     print(letter)
5     ix += 1
```

This loop traverses the string and displays each letter on a line by itself. It uses `ix` for the index, which does not make it any clearer. The loop condition is `ix < len(fruit)`, so when `ix` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string. However, this code is a lot longer than it needs to be, and not very clear at all.

But we've previously seen how the `for` loop can easily iterate over the elements in a list and it can do so for strings as well:

```
1 word="Banana"
2 for letter in word:
3     print(letter)
```

Each time through the loop, the next character in the string is assigned to the variable `c`. The loop continues until no characters are left. Here we can see the expressive power the `for` loop gives us compared to the `while` loop when traversing a string.

The following example shows how to use concatenation and a `for` loop to generate an abecedarian series. Abecedarian refers to a series or list in which the elements appear in alphabetical order. For example, in Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

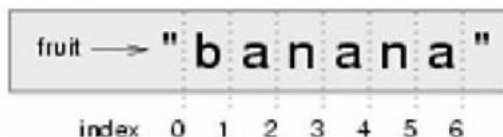
```
1 prefixes = "JKLMNOPQ"
2 suffix = "ack"
3
4 for p in prefixes:
5     print(p + suffix)
```

5.1.6 Slices

A *substring* of a string is obtained by taking a **slice**. Similarly, we can slice a list to refer to some sublist of the items in the list:

```
>>> phrase = "Pirates of the Caribbean"
>>> print(phrase[0:7])
Pirates
>>> print(phrase[11:14])
the
>>> print(phrase[13:24])
e Caribbean
>>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
>>> print(friends[2:4])
['Brad', 'Angelina']
```

The operator `[n:m]` returns the part of the string from the *n*'th character to the *m*'th character, including the first but excluding the last. This behavior makes sense if you imagine the indices pointing *between* the characters, as in the following diagram:



If you imagine this as a piece of paper, the slice operator `[n:m]` copies out the part of the paper between the *n* and *m* positions. Provided *m* and *n* are both within the bounds of the string, your result will be of length $(m-n)$.

Three tricks are added to this: if you omit the first index (before the colon), the slice starts at the beginning of the string (or list). If you omit the second index, the slice extends to the end of the string (or list). Similarly, if you provide value for *n* that is bigger than the length of the string (or list), the slice will take all the values up to the end. (It won't give an "out of range" error like the normal indexing operation does.) Thus:

```
>>> word = "banana"
>>> word[:3]
'ban'
>>> word[3:]
'ana'
>>> word[3:999]
'ana'
```

What do you think `phrase[:]` means? What about `friends[4:]`? `phrase[-5:-3]`?

5.1.7 String comparison

The comparison operators work on strings. To see if two strings are equal:

```
1 if word == "banana":
2     print("Yes, we have no bananas!")
```

Other comparison operations are useful for putting words in *lexicographical* order:

```
1 if word < "banana":
2     print("Your word, " + word + ", comes before banana.")
3 elif word > "banana":
4     print("Your word, " + word + ", comes after banana.")
5 else:
6     print("Yes, we have no bananas!")
```

This is similar to the alphabetical order you would use with a dictionary, except that all the uppercase letters come before all the lowercase letters. As a result:

```
Your word, Zebra, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. A more difficult problem is making the program realize that zebras are not fruit.

5.1.8 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:

```
1 greeting = "Hello, world!"
2 greeting[0] = 'J'           # ERROR!
3 print(greeting)
```

Instead of producing the output `Jello, world!`, this code produces the runtime error `TypeError: 'str' object does not support item assignment.`

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:

```
1 greeting = "Hello, world!"
2 new_greeting = "J" + greeting[1:]
3 print(new_greeting)
```

The solution here is to concatenate a new first letter onto a slice of `greeting`. This operation has no effect on the original string.

5.1.9 The `in` and `not in` operators

The `in` operator tests for membership. When both of the arguments to `in` are strings, `in` checks whether the left argument is a substring of the right argument.

```
>>> "p" in "apple"
True
>>> "i" in "apple"
False
```

(continues on next page)

(continued from previous page)

```
>>> "ap" in "apple"
True
>>> "pa" in "apple"
False
```

Note that a string is a substring of itself, and the empty string is a substring of any other string. (Also note that computer scientists like to think about these edge cases quite carefully!)

```
>>> "a" in "a"
True
>>> "apple" in "apple"
True
>>> "" in "a"
True
>>> "" in "apple"
True
```

The `not in` operator returns the logical opposite results of `in`:

```
>>> "x" not in "apple"
True
```

Combining the `in` operator with string concatenation using `+`, we can write a function that removes all the vowels from a string:

```
1 def remove_vowels(phrase):
2     vowels = "aeiou"
3     string_sans_vowels = ""
4     for letter in phrase:
5         if letter.lower() not in vowels:
6             string_sans_vowels += letter
7     return string_sans_vowels
```

Important to note is the `letter.lower()` in line 5, without it, any uppercase vowels would not be removed.

5.1.10 A find function

What does the following function do?

```
1 def my_find(haystack, needle):
2     """
3     Find and return the index of needle in haystack.
4     Return -1 if needle does not occur in haystack.
5     """
6     for index, letter in enumerate(haystack):
7         if letter == needle:
8             return index
9     return -1
```

Compare the output of the code above with what Python does itself with the code below:

```
1 haystack = "Bananarama!"
2 print(haystack.find('a'))
3 print(my_find(haystack, 'a'))
```

In a sense, `find` is the opposite of the indexing 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`.

This is another example where we see a `return` statement inside a loop. If `letter == needle`, the function returns immediately, breaking out of the loop prematurely.

If the character doesn't appear in the string, then the program exits the loop normally and returns `-1`.

This pattern of computation is sometimes called a **eureka traversal** or **short-circuit evaluation**, because as soon as we find what we are looking for, we can cry "Eureka!", take the short-circuit, and stop looking.

5.1.11 Looping and counting

The following program counts the number of times the letter `a` appears in a string, and is another example of the counter pattern introduced in *Counting digits*:

```
1 def count_a(text):
2     count = 0
3     for letter in text:
4         if letter == "a":
5             count += 1
6     return count
7
8 print(count_a("banana") == 3)
```

5.1.12 Optional parameters

To find the locations of the second or third occurrence of a character in a string, we can modify the `find` function, adding a third parameter for the starting position in the search string:

```
1 def find2(haystack, needle, start):
2     for index, letter in enumerate(haystack[start:]):
3         if letter == needle:
4             return index + start
5     return -1
6
7
8
9 print(find2("banana", "a", 2) == 3)
```

The call `find2("banana", "a", 2)` now returns `3`, the index of the first occurrence of `"a"` in `"banana"` starting the search at index `2`. What does `find2("banana", "n", 3)` return? If you said, `4`, there is a good chance you understand how `find2` works.

Better still, we can combine `find` and `find2` using an **optional parameter**:

```
1 def find(haystack, needle, start=0):
2     for index, letter in enumerate(haystack[start:]):
3         if letter == needle:
4             return index + start
5     return -1
```

When a function has an optional parameter, the caller *may* provide a matching argument. If the third argument is provided to `find`, it gets assigned to `start`. But if the caller leaves the argument out, then `start` is given a default value indicated by the assignment `start=0` in the function definition.

So the call `find("banana", "a", 2)` to this version of `find` behaves just like `find2`, while in the call `find("banana", "a")`, `start` will be set to the **default value** of 0.

Adding another optional parameter to `find` makes it search from a starting position, up to but not including the end position:

```
1 def find(haystack, needle, start=0, end=-1):
2     for index, letter in enumerate(haystack[start:end]):
3         if letter == needle:
4             return index + start
5     return -1
```

The semantics of `start` and `end` in this function are precisely the same as they are in the `range` function.

5.1.13 The built-in `find` method

Now that we've done all this work to write a powerful `find` function, we can reveal that strings already have their own built-in `find` method. It can do everything that our code can do, and more! Try all the examples listed above, and check the results!

The built-in `find` method is more general than our version. It can find substrings, not just single characters:

```
>>> "banana".find("nan")
2
>>> "banana".find("na", 3)
4
```

Usually we'd prefer to use the methods that Python provides rather than reinvent our own equivalents. But many of the built-in functions and methods make good teaching exercises, and the underlying techniques you learn are your building blocks to becoming a proficient programmer.

5.1.14 The `split` method

One of the most useful methods on strings is the `split` method: it splits a single multi-word string into a list of individual words, removing all the whitespace between them. (Whitespace means any tabs, newlines, or spaces.) This allows us to read input as a single string, and split it into words.

```
>>> phrase = "Well I never did said Alice"
>>> words = phrase.split()
>>> words
['Well', 'I', 'never', 'did', 'said', 'Alice']
```

5.1.15 Cleaning up your strings

We'll often work with strings that contain punctuation, or tab and newline characters, especially, as we'll see in a future chapter, when we read our text from files or from the Internet. But if we're writing a program, say, to count word frequencies or check the spelling of each word, we'd prefer to strip off these unwanted characters.

We'll show just one example of how to strip punctuation from a string. Remember that strings are immutable, so we cannot change the string with the punctuation — we need to traverse the original string and create a new string, omitting any punctuation:

```
1 punctuation = "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~"
2
3 def remove_punctuation(phrase):
4     phrase_sans_punct = ""
5     for letter in phrase:
6         if letter not in punctuation:
7             phrase_sans_punct += letter
8     return phrase_sans_punct
```

Setting up that first assignment is messy and error-prone. Fortunately, the Python `string` module already does it for us. So we will make a slight improvement to this program — we'll import the `string` module and use its definition:

```
1 import string
2
3 def remove_punctuation(phrase):
4     phrase_sans_punct = ""
5     for letter in phrase:
6         if letter not in string.punctuation:
7             phrase_sans_punct += letter
8     return phrase_sans_punct
```

5.1.16 The string format method

The easiest and most powerful way to format a string in Python 3 is to use the `format` method. To see how this works, let's start with a few examples:

```
1 phrase = "His name is {0}!".format("Arthur")
2 print(phrase)
3
4 name = "Alice"
```

(continues on next page)

(continued from previous page)

```
5 age = 10
6 phrase = "I am {1} and I am {0} years old.".format(age, name)
7 print(phrase)
8 phrase = "I am {0} and I am {1} years old.".format(age, name)
9 print(phrase)
10
11 x = 4
12 y = 5
13 phrase = "2**10 = {0} and {1} * {2} = {3:f}".format(2**10, x, y, x * y)
14 print(phrase)
```

Running the script produces:

```
His name is Arthur!
I am Alice and I am 10 years old.
I am 10 and I am Alice years old.
2**10 = 1024 and 4 * 5 = 20.000000
```

The template string contains *place holders*, ... `{0}` ... `{1}` ... `{2}` ... etc. The `format` method substitutes its arguments into the place holders. The numbers in the place holders are indexes that determine which argument gets substituted — make sure you understand line 6 above!

But there's more! Each of the replacement fields can also contain a **format specification** — it is always introduced by the `:` symbol (Line 13 above uses one.) This modifies how the substitutions are made into the template, and can control things like:

- whether the field is aligned to the left `<`, center `^`, or right `>`
- the width allocated to the field within the result string (a number like 10)
- the type of conversion (we'll initially only force conversion to float, `f`, as we did in line 13 of the code above, or perhaps we'll ask integer numbers to be converted to hexadecimal using `x`)
- if the type conversion is a float, you can also specify how many decimal places are wanted (typically, `.2f` is useful for working with currencies to two decimal places.)

Let's do a few simple and common examples that should be enough for most needs. If you need to do anything more esoteric, use `help` and read all the powerful, gory details.

```
1 name1 = "Paris"
2 name2 = "Whitney"
3 name3 = "Hilton"
4
5 print("Pi to three decimal places is {0:.3f}".format(3.1415926))
6 print("123456789 123456789 123456789 123456789 123456789 123456789")
7 print("|||{0:<15}|||{1:^15}|||{2:>15}|||Born in {3}|||"
8       .format(name1, name2, name3, 1981))
9 print("The decimal value {0} converts to hex value {0:x}"
10       .format(123456))
```

This script produces the output:

```

1 letter = """
2 Dear {0} {2}.
3 {0}, I have an interesting money-making proposition for you!
4 If you deposit $10 million into my bank account, I can
5 double your money ...
6 """
7
8 print(letter.format("Paris", "Whitney", "Hilton"))
9 print(letter.format("Bill", "Henry", "Gates"))

```

This produces the following:

```

Dear Paris Hilton.
Paris, I have an interesting money-making proposition for you!
If you deposit $10 million into my bank account, I can
double your money ...

Dear Bill Gates.
Bill, I have an interesting money-making proposition for you!
If you deposit $10 million into my bank account I can
double your money ...

```

As you might expect, you'll get an index error if your placeholders refer to arguments that you do not provide:

```

>>> "hello {3}".format("Dave")
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
IndexError: tuple index out of range

```

The following example illustrates the real utility of string formatting. First, we'll try to print a table without using string formatting:

```

1 print("i\ti**2\ti**3\ti**5\ti**10\ti**20")
2 for i in range(1, 11):
3     print(i, "\t", i**2, "\t", i**3, "\t", i**5, "\t",
4           i**10, "\t", i**20)

```

This script produces the output:

```

Pi to three decimal places is 3.142
123456789 123456789 123456789 123456789 123456789 123456789
|||Paris    |||    Whitney    |||    Hilton|||Born in 1981|||
The decimal value 123456 converts to hex value 1e240

```

5.2 Tuples

5.2.1 Tuples are used for grouping data

We saw earlier that we could group together pairs of values by surrounding with parentheses. Recall this example:

```
>>> year_born = ("Paris Hilton", 1981)
```

This is an example of a **data structure** — a mechanism for grouping and organizing data to make it easier to use.

The pair is an example of a **tuple**. Generalizing this, a tuple can be used to group any number of items into a single compound value. Syntactically, a tuple is a comma-separated sequence of values. Although it is not necessary, it is conventional to enclose tuples in parentheses:

```
>>> julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress",  
→ "Atlanta, Georgia")
```

The other thing that could be said somewhere around here, is that the parentheses are there to disambiguate. For example, if we have a tuple nested within another tuple and the parentheses weren't there, how would we tell where the nested tuple begins/ends? Also: the creation of an empty tuple is done like this: `empty_tuple=()`

Tuples are useful for representing what other languages often call *records* (or structs) — some related information that belongs together, like your student record. There is no description of what each of these fields means, but we can guess. A tuple lets us “chunk” together related information and use it as a single thing.

Tuples support the same sequence operations as strings. The index operator selects an element from a tuple.

```
>>> julia[2]
1967
```

But if we try to use item assignment to modify one of the elements of the tuple, we get an error:

```
>>> julia[0] = "X"
TypeError: 'tuple' object does not support item assignment
```

So like strings, tuples are immutable. Once Python has created a tuple in memory, it cannot be changed.

Of course, even if we can't modify the elements of a tuple, we can always make the `julia` variable reference a new tuple holding different information. To construct the new tuple, it is convenient that we can slice parts of the old tuple and join up the bits to make the new tuple. So if `julia` has a new recent film, we could change her variable to reference a new tuple that used some information from the old one:

```
>>> julia = julia[:3] + ("Eat Pray Love", 2010) + julia[5:]
>>> julia
('Julia', 'Roberts', 1967, 'Eat Pray Love', 2010, 'Actress', 'Atlanta, GA', 'Georgia')
```

To create a tuple with a single element (but you're probably not likely to do that too often), we have to include the final comma, because without the final comma, Python treats the `(5)` below as an integer in parentheses:

```
>>> tup = (5,)
>>> type(tup)
<class 'tuple'>
>>> x = (5)
>>> type(x)
<class 'int'>
```

5.2.2 Tuple assignment

Python has a very powerful **tuple assignment** feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment. (We already saw this used for pairs, but it generalizes.)

```
(name, surname, year_born, movie, year_movie, profession, birthplace) = julia
```

This does the equivalent of seven assignment statements, all on one easy line. One requirement is that the number of variables on the left must match the number of elements in the tuple.

One way to think of tuple assignment is as tuple packing/unpacking.

In tuple packing, the values on the left are ‘packed’ together in a tuple:

```
>>> bob = ("Bob", 19, "CS")    # tuple packing
```

In tuple unpacking, the values in a tuple on the right are ‘unpacked’ into the variables/names on the right:

```
>>> bob = ("Bob", 19, "CS")
>>> (name, age, studies) = bob    # tuple unpacking
>>> name
```

(continues on next page)

Once in a while, it is useful to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap `a` and `b`:

```
'Bob'  
>>> age  
19  
>>> studies  
'CS'
```

```
1 temp = a  
2 a = b  
3 b = temp
```

Tuple assignment solves this problem neatly:

```
1 (a, b) = (b, a)
```

The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile.

Naturally, the number of variables on the left and the number of values on the right have to be the same:

```
>>> (one, two, three, four) = (1, 2, 3)  
ValueError: need more than 3 values to unpack
```

5.2.3 Tuples as return values

Functions can always only return a single value, but by making that value a tuple, we can effectively group together as many values as we like, and return them together. This is very useful — we often want to know some batsman's highest and lowest score, or we want to find the mean and the standard deviation, or we want to know the year, the month, and the day, or if we're doing some ecological modelling we may want to know the number of rabbits and the number of wolves on an island at a given time.

For example, we could write a function that returns both the area and the circumference of a circle of radius r :

```
1 def circle_stats(r):  
2     """ Return (circumference, area) of a circle of radius r """  
3     circumference = 2 * math.pi * r  
4     area = math.pi * r * r  
5     return (circumference, area)
```

5.2.4 Composability of Data Structures

We saw in an earlier chapter that we could make a list of pairs, and we had an example where one of the items in the tuple was itself a list:

```
students = [  
    ("John", ["CompSci", "Physics"]),  
    ("Vusi", ["Maths", "CompSci", "Stats"]),  
    ("Jess", ["CompSci", "Accounting", "Economics", "Management"]),  
    ("Sarah", ["InfSys", "Accounting", "Economics", "CommLaw"]),  
    ("Zuki", ["Sociology", "Economics", "Law", "Stats", "Music"])]
```

Tuples items can themselves be other tuples. For example, we could improve the information about our movie stars to hold the full date of birth rather than just the year, and we could have a list of some of her movies and dates that they were made, and so on:

```
julia_more_info = ( ("Julia", "Roberts"), (8, "October", 1967),  
                    "Actress", ("Atlanta", "Georgia"),  
                    [ ("Duplicity", 2009),  
                      ("Notting Hill", 1999),  
                      ("Pretty Woman", 1990),  
                      ("Erin Brockovich", 2000),  
                      ("Eat Pray Love", 2010),  
                      ("Mona Lisa Smile", 2003),  
                      ("Oceans Twelve", 2004) ])
```

Notice in this case that the tuple has just five elements — but each of those in turn can be another tuple, a list, a string, or any other kind of Python value. This property is known as being **heterogeneous**, meaning that it can be composed of elements of different types.

5.2.5 Glossary

data structure An organization of data for the purpose of making it easier to use.

immutable data value A data value which cannot be modified. Assignments to elements or slices (sub-parts) of immutable values cause a runtime error.

mutable data value A data value which can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.

tuple An immutable data value that contains related elements. Tuples are used to group together related data, such as a person's name, their age, and their gender.

tuple assignment An assignment to all of the elements in a tuple using a single assignment statement. Tuple assignment occurs *simultaneously* rather than in sequence, making it useful for swapping values.

5.3 Lists

A **list** is an **ordered** collection of values. The values that make up a list are called its **elements**, or its **items**. We will use the term *element* or *item* to mean the same thing. Lists are similar to strings, which are ordered collections of characters, except that the elements of a list can be of any type. Lists and strings — and other collections that maintain the order of their items — are called **sequences**.

5.3.1 List values

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

```
1 numbers = [10, 20, 30, 40]
2 words = ["spam", "bungee", "swallow"]
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (amazingly) another list:

```
1 stuffs = ["hello", 2.0, 5, [10, 20]]
```

A list within another list is said to be **nested**.

Finally, a list with no elements is called an **empty list**, and is denoted [].

We have already seen that we can assign list values to variables or pass lists as parameters to functions:

```
1 >>> vocabulary = ["apple", "cheese", "dog"]
2 >>> numbers = [17, 123]
3 >>> an_empty_list = []
4 >>> print(vocabulary, numbers, an_empty_list)
5 ["apple", "cheese", "dog"] [17, 123] []
```

5.3.2 Accessing elements

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string — the index operator: [] (not to be confused with an empty list). The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> numbers[0]
17
```

Any expression evaluating to an integer can be used as an index:

```
>>> numbers[9-8]
123
>>> numbers[1.0]
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: list indices must be integers, not float
```

If you try to access or assign to an element that does not exist, you get a runtime error:

```
>>> numbers[2]
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
IndexError: list index out of range
```

It is common (but wrong!) to use a loop variable as a list index.

```
1 horsemen = ["war", "famine", "pestilence", "death"]
2
3 for i in [0, 1, 2, 3]:
4     print(horsemen[i])
```

Each time through the loop, the variable `i` is used as an index into the list, printing the `i`'th element. This pattern of computation is called a **list traversal**.

The above sample doesn't need or use the index `i` for anything besides getting the items from the list, so this more direct version — where the `for` loop gets the items — is much more clear!

```
1 horsemen = ["war", "famine", "pestilence", "death"]
2
3 for h in horsemen:
4     print(h)
```

5.3.3 List length

The function `len` returns the length of a list, which is equal to the number of its elements. If you are going to use an integer index to access the list, it is a good idea to use this value as the upper bound of a loop instead of a constant. That way, if the size of the list changes, you won't have to go through the program changing all the loops; they will work correctly for any size list:

```
1 horsemen = ["war", "famine", "pestilence", "death"]
2
3 for i in range(len(horsemen)):
4     print(horsemen[i])
```

The last time the body of the loop is executed, `i` is `len(horsemen) - 1`, which is the index of the last element. (But the version without the index looks even better now! The version above is not the right way to do things!)

```
1 horsemen = ["war", "famine", "pestilence", "death"]
2
3 for horseman in horsemen:
4     print(horseman)
```

Although a list can contain another list, the nested list still counts as a single element in its parent list. The length of this list is 4:

```
>>> len(["car makers", 1, ["Ford", "Toyota", "BMW"], [1, 2, 3]])
4
```

5.3.4 List membership

`in` and `not in` are Boolean operators that test membership in a sequence. We used them previously with strings, but they also work with lists and other sequences:

```
>>> horsemen = ["war", "famine", "pestilence", "death"]
>>> "pestilence" in horsemen
True
>>> "debauchery" in horsemen
False
>>> "debauchery" not in horsemen
True
```

Using this produces a more elegant version of the nested loop program we previously used to count the number of students doing Computer Science in the section *Nested Loops for Nested Data*:

```
1 students = [  
2     ("John", ["CompSci", "Physics"]),  
3     ("Vusi", ["Maths", "CompSci", "Stats"]),  
4     ("Jess", ["CompSci", "Accounting", "Economics", "Management"]),  
5     ("Sarah", ["InfSys", "Accounting", "Economics", "CommLaw"]),  
6     ("Zuki", ["Sociology", "Economics", "Law", "Stats", "Music"])]
```

(continues on next page)

(continued from previous page)

```
7  
8 # Count how many students are taking CompSci  
9 counter = 0  
10 for name, subjects in students:  
11     if "CompSci" in subjects:  
12         counter += 1  
13  
14 print("The number of students taking CompSci is", counter)
```

5.3.5 List operations

The + operator concatenates lists:

```
>>> first_list = [1, 2, 3]  
>>> second_list = [4, 5, 6]  
>>> both_lists = first_list + second_list  
>>> both_lists  
[1, 2, 3, 4, 5, 6]
```

Similarly, the * operator repeats a list a given number of times:

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

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

5.3.6 List slices

The slice operations we saw previously with strings let us work with sublists:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]  
>>> a_list[1:3]  
['b', 'c']  
>>> a_list[:4]  
['a', 'b', 'c', 'd']  
>>> a_list[3:]  
['d', 'e', 'f']  
>>> a_list[:]  
['a', 'b', 'c', 'd', 'e', 'f']
```

5.3.7 Lists are mutable

Unlike strings, lists are **mutable**, which means we can change their elements. Using the index operator on the left side of an assignment, we can update one of the elements:

```
>>> fruit = ["banana", "apple", "quince"]
>>> fruit[0] = "pear"
>>> fruit[2] = "orange"
>>> fruit
['pear', 'apple', 'orange']
```

The bracket operator applied to a list can appear anywhere in an expression. When it appears on the left side of an assignment, it changes one of the elements in the list, so the first element of `fruit` has been changed from "banana" to "pear", and the last from "quince" to "orange". An assignment to an element of a list is called **item assignment**. Item assignment does not work for strings:

```
>>> my_string = "TEST"
>>> my_string[2] = "X"
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

but it does for lists:

```
>>> my_list = ["T", "E", "S", "T"]
>>> my_list[2] = "X"
>>> my_list
['T', 'E', 'X', 'T']
```

With the slice operator we can update a whole sublist at once:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3] = ["x", "y"]
>>> a_list
['a', 'x', 'y', 'd', 'e', 'f']
```

We can also remove elements from a list by assigning an empty list to them:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3] = []
>>> a_list
['a', 'd', 'e', 'f']
```

And we can add elements to a list by squeezing them into an empty slice at the desired location:

```
>>> a_list = ["a", "d", "f"]
>>> a_list[1:1] = ["b", "c"]
>>> a_list
['a', 'b', 'c', 'd', 'f']
>>> a_list[4:4] = ["e"]
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f']
```

5.3.8 List deletion

Using slices to delete list elements can be error-prone. Python provides an alternative that is more readable. The `del` statement removes an element from a list:

```
>>> a = ["one", "two", "three"]
>>> del a[1]
>>> a
['one', 'three']
```

As you might expect, `del` causes a runtime error if the index is out of range.

You can also use `del` with a slice to delete a sublist:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> del a_list[1:5]
>>> a_list
['a', 'f']
```

As usual, the sublist selected by slice contains all the elements up to, but not including, the second index.

5.3.9 Objects and references

After we execute these assignment statements

```
1 a = "banana"
2 b = "banana"
```

we know that `a` and `b` will refer to a string object with the letters "banana". But we don't know yet whether they point to the *same* string object.

There are two possible ways the Python interpreter could arrange its memory:



In one case, `a` and `b` refer to two different objects that have the same value. In the second case, they refer to the same object.

We can test whether two names refer to the same object using the `is` operator:

```
>>> a is b
True
```

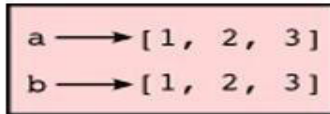
This tells us that both `a` and `b` refer to the same object, and that it is the second of the two state snapshots that accurately describes the relationship.

Since strings are *immutable*, Python optimizes resources by making two names that refer to the same string value refer to the same object.

This is not the case with lists:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
>>> a is b
False
```

The state snapshot here looks like this:



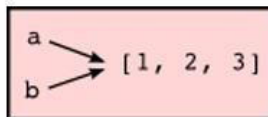
`a` and `b` have the same value but do not refer to the same object.

5.3.10 Aliasing

Since variables refer to objects, if we assign one variable to another, both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
```

In this case, the state snapshot looks like this:



Because the same list has two different names, `a` and `b`, we say that it is **aliased**. Changes made with one alias affect the other:

```
>>> b[0] = 5
>>> a
[5, 2, 3]
```

Although this behavior can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects (i.e. lists at this point in our textbook, but we'll meet more mutable objects as we cover classes and objects, dictionaries and sets). Of course, for immutable objects (i.e. strings, tuples), there's no problem — it is just not possible to change something and get a surprise when you access an alias name. That's why Python is free to alias strings (and any other immutable kinds of data) when it sees an opportunity to economize.

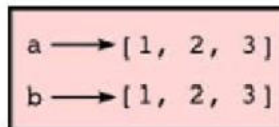
5.3.11 Cloning lists

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called **cloning**, to avoid the ambiguity of the word copy.

The easiest way to clone a list is to use the slice operator:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
```

Taking any slice of `a` creates a new list. In this case the slice happens to consist of the whole list. So now the relationship is like this:



Now we are free to make changes to `b` without worrying that we'll inadvertently be changing `a`:

```
>>> b[0] = 5
>>> a
[1, 2, 3]
```

5.3.12 Lists and for loops

The `for` loop also works with lists, as we've already seen. The generalized syntax of a `for` loop is:

```
for <VARIABLE> in <LIST>:
    <BODY>
```

So, as we've seen

```
1 friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
2 for friend in friends:
3     print(friend)
```

It almost reads like English: For (every) friend in (the list of) friends, print (the name of the) friend.

Any list expression can be used in a `for` loop:

```
1 for number in range(20):
2     if number % 3 == 0:
3         print(number)
4
5 for fruit in ["banana", "apple", "quince"]:
6     print("I like to eat " + fruit + "s!")
```

The first example prints all the multiples of 3 between 0 and 19. The second example expresses enthusiasm for various fruits.

Since lists are mutable, we often want to traverse a list, changing each of its elements. The following squares all the numbers in the list `xs`:

```
1 xs = [1, 2, 3, 4, 5]
2
3 for i in range(len(xs)):
4     xs[i] = xs[i]**2
```

Take a moment to think about `range(len(xs))` until you understand how it works.

In this example we are interested in both the *value* of an item, (we want to square that value), and its *index* (so that we can assign the new value to that position). This pattern is common enough that Python provides a nicer way to implement it:

```
1 xs = [1, 2, 3, 4, 5]
2
3 for (i, val) in enumerate(xs):
4     xs[i] = val**2
```

`enumerate` generates pairs of both (index, value) during the list traversal. Try this next example to see more clearly how `enumerate` works:

```
1 for (i, v) in enumerate(["banana", "apple", "pear", "lemon"]):
2     print(i, v)
```

```
0 banana
1 apple
2 pear
3 lemon
```

5.3.13 List parameters

Passing a list as an argument actually passes a reference to the list, not a copy or clone of the list. So parameter passing creates an alias for you: the caller has one variable referencing the list, and the called function has an alias, but there is only one underlying list object. For example, the function below takes a list as an argument and multiplies each element in the list by 2:

```
1 def double_stuff(stuff_list):
2     """ Overwrite each element in a list with double its value. """
3     for (index, stuff) in enumerate(stuff_list):
4         stuff_list[index] = 2 * stuff
```

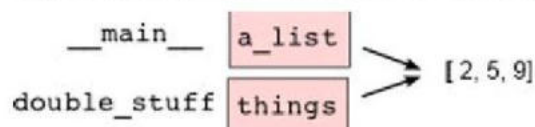
If we add the following onto our script:

```
1 things = [2, 5, 9]
2 double_stuff(things)
3 print(things)
```

When we run it we'll get:

```
[4, 10, 18]
```

In the function above, the parameter `stuff_list` and the variable `things` are aliases for the same object. So before any changes to the elements in the list, the state snapshot looks like this:



Since the list object is shared by two frames, we drew it between them.

If a function modifies the items of a list parameter, the caller sees the change.

5.3.14 List methods

The dot operator can also be used to access built-in methods of list objects. We'll start with the most useful method for adding something onto the end of an existing list:

```
>>> mylist = []
>>> mylist.append(5)
>>> mylist.append(27)
>>> mylist.append(3)
>>> mylist.append(12)
>>> mylist
[5, 27, 3, 12]
```

`append` is a list method which adds the argument passed to it to the end of the list. We'll use it heavily when we're creating new lists. Continuing with this example, we show several other list methods:

```
>>> mylist.insert(1, 12)  # Insert 12 at pos 1, shift other items up
>>> mylist
[5, 12, 27, 3, 12]
>>> mylist.count(12)      # How many times is 12 in mylist?
2
>>> mylist.extend([5, 9, 5, 11])  # Put whole list onto end of mylist
```

(continues on next page)
(continued from previous page)

```
>>> mylist
[5, 12, 27, 3, 12, 5, 9, 5, 11]
>>> mylist.index(9)      # Find index of first 9 in mylist
6
>>> mylist.reverse()
>>> mylist
[11, 5, 9, 5, 12, 3, 27, 12, 5]
>>> mylist.sort()
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 12, 27]
>>> mylist.remove(12)    # Remove the first 12 in the list
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 27]
```

Experiment and play with the list methods shown here, and read their documentation until you feel confident that you understand how they work.

5.3.15 Pure functions and modifiers

As seen before, there is a difference between a pure function and one with side-effects. The difference is shown below as lists have some special gotcha's. Functions which take lists as arguments and change them during execution are called **modifiers** and the changes they make are called **side effects**.

A **pure function** does not produce side effects. It communicates with the calling program only through parameters, which it does not modify, and a return value. Here is `double_stuff` written as a pure function:

```
1 def double_stuff(a_list):
2     """ Return a new list which contains
3         doubles of the elements in a_list.
4     """
5     new_list = []
6     for value in a_list:
7         new_elem = 2 * value
8         new_list.append(new_elem)
9
10    return new_list
```

This version of `double_stuff` does not change its arguments:

```
>>> things = [2, 5, 9]
>>> more_things = double_stuff(things)
>>> things
[2, 5, 9]
>>> more_things
[4, 10, 18]
```

An early rule we saw for assignment said "first evaluate the right hand side, then assign the resulting value to the variable". So it is quite safe to assign the function result to the same variable that was passed to the function:

```
>>> things = [2, 5, 9]
>>> things = double_stuff(things)
>>> things
[4, 10, 18]
```

5.3.16 Functions that produce lists

The pure version of `double_stuff` above made use of an important **pattern** for your toolbox. Whenever you need to write a function that creates and returns a list, the pattern is usually:

```
1 initialize a result variable to be an empty list
2 loop
3     create a new element
4     append it to result
5 return the result
```

Let us show another use of this pattern. Assume you already have a function `is_prime(x)` that can test if `x` is prime. Write a function to return a list of all prime numbers less than `n`:

```
1 def primes_less_than(n):
2     """ Return a list of all prime numbers less than n. """
3     result = []
4     for i in range(2, n):
5         if is_prime(i):
6             result.append(i)
7     return result
```

5.3.17 Strings and lists

Two of the most useful methods on strings involve conversion to and from lists of substrings. The `split` method (which we've already seen) breaks a string into a list of words. By default, any number of whitespace characters is considered a word boundary:

```
>>> song = "The rain in Spain..."
>>> words = song.split()
>>> words
['The', 'rain', 'in', 'Spain...']
```

An optional argument called a **delimiter** can be used to specify which string to use as the boundary marker between substrings. The following example uses the string `ai` as the delimiter:

```
>>> song.split("ai")
['The r', 'n in Sp', 'n...']
```

Notice that the delimiter doesn't appear in the result.

The inverse of the `split` method is `join`. You choose a desired **separator** string, (often called the *glue*) and join the list with the glue between each of the elements:

Notice that the delimiter doesn't appear in the result.

The inverse of the `split` method is `join`. You choose a desired **separator** string, (often called the *glue*) and join the list with the glue between each of the elements:

```
>>> glue = ";"
>>> phrase = glue.join(words)
>>> phrase
'The;rain;in;Spain...'
```

The list that you glue together (words in this example) is not modified. Also, as these next examples show, you can use empty glue or multi-character strings as glue:

```
>>> " --- ".join(words)
'The --- rain --- in --- Spain...'
>>> "".join(words)
'TheraininSpain...'
```

5.3.18 list and range

Python has a built-in type conversion function called `list` that tries to turn whatever you give it into a list.

```
>>> letters = list("Crunchy Frog")
>>> letters
['C', 'r', 'u', 'n', 'c', 'h', 'y', ' ', 'F', 'r', 'o', 'g']
>>> "".join(letters)
'Crunchy Frog'
```

One particular feature of `range` is that it doesn't instantly compute all its values: it "puts off" the computation, and does it on demand, or "lazily". We'll say that it gives a **promise** to produce the values when they are needed. This is very convenient if your computation short-circuits a search and returns early, as in this case:

```
1 def f(n):
2     """ Find the first positive integer between 101 and less
3         than n that is divisible by 21
4     """
5     for i in range(101, n):
6         if (i % 21 == 0):
7             return i
8
9
10 print(f(110) == 105)
11 print(f(1000000000) == 105)
```

In the second test, if `range` were to eagerly go about building a list with all those elements, you would soon exhaust your computer's available memory and crash the program. But it is cleverer than that! This computation works just fine, because the `range` object is just a promise to produce the elements if and when they are needed. Once the condition in the `if` becomes true, no further elements are generated, and the function returns. (Note: Before Python 3, `range` was not lazy. If you use an earlier versions of Python, YMMV!)

YMMV: Your Mileage May Vary

The acronym YMMV stands for *your mileage may vary*. American car advertisements often quoted fuel consumption figures for cars, e.g. that they would get 28 miles per gallon. But this always had to be accompanied by legal small-print warning the reader that they might not get the same. The term YMMV is now used idiomatically to mean "your results may differ", e.g. *The battery life on this phone is 3 days, but YMMV*.

You'll sometimes find the lazy `range` wrapped in a call to `list`. This forces Python to turn the lazy promise into an actual list:

```
>>> range(10)           # Create a lazy promise
range(0, 10)
>>> list(range(10))     # Call in the promise, to produce a list.
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

5.3.19 Looping and lists

Computers are useful because they can repeat computation, accurately and fast. So loops are going to be a central feature of almost all programs you encounter.

generate them.

Here are two functions that both generate ten million random numbers, and return the sum of the numbers. They both work.

```

1 import random
2 joe = random.Random()
3
4 def sum1():
5     """ Build a list of random numbers, then sum them """
6     xs = []
7     for i in range(10000000):
8         num = joe.randrange(1000) # Generate one random number
9         xs.append(num)           # Save it in our list
10
11     tot = sum(xs)
12     return tot
13
14 def sum2():
15     """ Sum the random numbers as we generate them """
16     tot = 0
17     for i in range(10000000):
18         num = joe.randrange(1000)
19         tot += num
20     return tot
21
22 print(sum1())
23 print(sum2())

```

What reasons are there for preferring the second version here? (Hint: open a tool like the Performance Monitor on your computer, and watch the memory usage. How big can you make the list before you get a fatal memory error in `sum1`?)

In a similar way, when working with files, we often have an option to read the whole file contents into a single string, or we can read one line at a time and process each line as we read it. Line-at-a-time is the more traditional and perhaps safer way to do things — you'll be able to work comfortably no matter how large the file is. (And, of course, this mode of processing the files was essential in the old days when computer memories were much smaller.) But you may find whole-file-at-once is sometimes more convenient!

5.3.20 Nested lists

A nested list is a list that appears as an element in another list. In this list, the element with index 3 is a nested list:

```
>>> nested = ["hello", 2.0, 5, [10, 20]]
```

If we output the element at index 3, we get:

```
>>> print(nested[3])
[10, 20]
```

To extract an element from the nested list, we can proceed in two steps:

```
>>> elem = nested[3]
>>> elem[0]
10
```

Or we can combine them:

```
>>> nested[3][1]
20
```

Bracket operators evaluate from left to right, so this expression gets the 3'th element of `nested` and extracts the 1'th element from it.

5.3.21 Matrices

Nested lists are often used to represent matrices. For example, the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

might be represented as:

```
>>> mx = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`mx` is a list with three elements, where each element is a row of the matrix. We can select an entire row from the matrix in the usual way:

```
>>> mx[1]
[4, 5, 6]
```

Or we can extract a single element from the matrix using the double-index form:

```
>>> mx[1][2]
6
```

The first index selects the row, and the second index selects the column. Although this way of representing matrices is common, it is not the only possibility. A small variation is to use a list of columns instead of a list of rows. Later we will see a more radical alternative using a dictionary.