Python: 4th lesson - Lists

Lists:

* Lists in Python represent ordered sequences of values. Here is an example of how to create them:

primes = [2, 3, 5, 7]

We can put other types of things in lists:

planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

We can even make a list of lists:

hands = [

['J', 'Q', 'K'],

['2', '2', '2'],

['6', 'A', 'K'], *# (Comma after the last element is optional)*

]

*# (I could also have written this on one line, but it can get hard to read)*

hands = [['J', 'Q', 'K'], ['2', '2', '2'], ['6', 'A', 'K']]

* A list can contain a mix of different types of variables:

my\_favourite\_things = [32, 'raindrops on roses', help]

*# (Yes, Python's help function is \*definitely\* one of my favourite things)*

Indexing:

The user can access individual list elements with square-brackets.

* Which planet is closest to the sun? Python uses *zero-based indexing*, so the first element has the index 0.

planets[0]

'Mercury'

* What’s the next closest planet?

planets[1]

'Venus'

* Which planet is furthest from the sun?

(Elements at the end of the list can be accessed with negative numbers, starting from -1)

planets[-1]

'Neptune'

planets[-2]

'Uranus'

Slicing:

* What are the first three planets? We can answer this question using slicing:

planets[0:3]

['Mercury', 'Venus', 'Earth']

planets[0:3] is our way of asking for the elements of planets starting from index 0 and continuing up to but not including index 3.

* The starting and ending indices are both optional. If leaving out the start index, it's assumed to be 0. So it could be rewriting the expression above as:

planets[:3]

['Mercury', 'Venus', 'Earth']

If leaving out the end index, it's assumed to be the length of the list.

planets[3:]

['Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

Which means, the expression above means "give me all the planets from index 3 onward".

We can also use negative indices when slicing:

*# All the planets except the first and last*

planets[1:-1]

['Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus']

*# The last 3 planets*

planets[-3:]

['Saturn', 'Uranus', 'Neptune']

Changing lists:

Lists are "mutable", meaning they can be modified "in place". One way to modify a list is to assign to an index or slice expression. For example, let's say we want to rename Mars:

planets[3] = 'Malacandra'

planets

['Mercury',

'Venus',

'Earth',

'Malacandra',

'Jupiter',

'Saturn',

'Uranus',

'Neptune']

Hm, that's quite a mouthful. Let's compensate by shortening the names of the first 3 planets.

planets[:3] = ['Mur', 'Vee', 'Ur']

print(planets)

*# That was silly. Let's give them back their old names*

planets[:4] = ['Mercury', 'Venus', 'Earth', 'Mars',]

['Mur', 'Vee', 'Ur', 'Malacandra', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

List functions:

Python has several useful functions for working with lists. len gives the length of a list:

*# How many planets are there?*

len(planets)

8

sorted returns a sorted version of a list:

*# The planets sorted in alphabetical order*

sorted(planets)

['Earth', 'Jupiter', 'Mars', 'Mercury', 'Neptune', 'Saturn', 'Uranus', 'Venus']

sum does what you might expect:

primes = [2, 3, 5, 7]

sum(primes)

17

We've previously used the min and max to get the minimum or maximum of several arguments. But we can also pass in a single list argument.

max(primes)

7

Objects:

We have used the term 'object' a lot so far - you may have even read that everything in Python is an object. What does that mean? In short, objects carry some things around with them. You access that stuff using Python's dot syntax. For example, numbers in Python carry around an associated variable called imag representing their imaginary part. (You'll probably never need to use this unless you're doing some very weird math.)

x = 12

*# x is a real number, so its imaginary part is 0.*

print(x.imag)

*# Here's how to make a complex number, in case you've ever been curious:*

c = 12 + 3j

print(c.imag)

0

3.0

The things an object carries around can also include functions. A function attached to an object is called a **method**. (Non-function things attached to an object, such as imag, are called attributes). For example, numbers have a method called bit\_length. Again, we access it using dot syntax:

x.bit\_length

<function int.bit\_length()>

To actually call it, we add parentheses:

x.bit\_length()

4

In the same way that we can pass functions to the help function (e.g. help(max)), we can also pass in methods:

help(x.bit\_length)

Help on built-in function bit\_length:

bit\_length() method of builtins.int instance

Number of bits necessary to represent self in binary.

>>> bin(37)

'0b100101'

>>> (37).bit\_length()

6

The examples above were utterly obscure. None of the types of objects we've looked at so far (numbers, functions, Booleans) have attributes or methods you're likely ever to use. But it turns out that lists have several methods which you'll use all the time.

List methods:

list.append modifies a list by adding an item to the end:

*# Pluto is a planet darn it!*

planets.append('Pluto')

Why does the cell above have no output? Let's check the documentation by calling help(planets.append).

help(planets.append)

Help on built-in function append:

append(object, /) method of builtins.list instance

Append object to the end of the list.

The -> None part is telling us that list.append doesn't return anything. But if we check the value of planets, we can see that the method call modified the value of planets:

planets

['Mercury',

'Venus',

'Earth',

'Mars',

'Jupiter',

'Saturn',

'Uranus',

'Neptune',

'Pluto']

list.pop removes and returns the last element of a list:

planets.pop()

'Pluto'

planets

['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

Searching lists:

Where does Earth fall in the order of planets? We can get its index using the list.index method.

planets.index('Earth')

2

It comes third (i.e. at index 2 - 0 indexing!). At what index does Pluto occur?

planets.index('Pluto')

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

/tmp/ipykernel\_19/2263615293.py in <module>

----> 1 planets.index('Pluto')

ValueError: 'Pluto' is not in list

To avoid unpleasant surprises like this, we can use the in operator to determine whether a list contains a particular value:

*# Is Earth a planet?*

"Earth" **in** planets

True

*# Is Calbefraques a planet?*

"Calbefraques" **in** planets

False

There are a few more interesting list methods we haven't covered. If you want to learn about all the methods and attributes attached to a particular object, we can call help() on the object itself. For example, help(planets) will tell us about all the list methods:

help(planets)

Tuples:

Tuples are almost exactly the same as lists. They differ in just two ways.

* The syntax for creating them uses parentheses instead of square brackets:

t = (1, 2, 3)

t = 1, 2, 3 *# equivalent to above*

t

(1, 2, 3)

* The values cannot be modified due to its immutability:

t[0] = 100

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

/tmp/ipykernel\_19/816329950.py in <module>

----> 1 t[0] = 100

TypeError: 'tuple' object does not support item assignment

* Tuples are often used for functions that have multiple return values. For example, the as\_integer\_ratio() method of float objects returns a numerator and a denominator in the form of a tuple:

x = 0.125

x.as\_integer\_ratio()

(1, 8)

* These multiple return values can be individually assigned as follows:

numerator, denominator = x.as\_integer\_ratio()

print(numerator / denominator)

0.125

* Finally we have some insight into the classic Stupid Python Trick™ for swapping two variables!

a = 1

b = 0

a, b = b, a

print(a, b)

0 1