Lists

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

In [1]:
```
primes = [2, 3, 5, 7]
```

We can put other types of things in lists:

In [2]:
```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus
```

We can even make a list of lists:

In [3]:
```
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:

In [4]:
```
my_favourite_things = [32, 'raindrops on roses', help]
# (Yes, Python's help function is *definitely* one of my favourite things)
```

Indexing

You 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 index 0.

In [5]:
```
planets[0]
```

Out[5]:
```
'Mercury'
```

What's the next closest planet?

In [6]:
```
planets[1]
```

Out[6]:
```
'Venus'
```

Which planet is *furthest* from the sun?

Loading [MathJax]/extensions/Safe.js

The end of the list can be accessed with negative numbers, starting from -1:

In [7]:
```
planets[-1]
```

Out[7]:
```
'Neptune'
```

In [8]:
```
planets[-2]
```

Out[8]:
```
'Uranus'
```

Slicing

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

In [9]:
```
planets[0:3]
```

Out[9]:
```
['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 I leave out the start index, it's assumed to be 0. So I could rewrite the expression above as:

In [10]:
```
planets[:3]
```

Out[10]:
```
['Mercury', 'Venus', 'Earth']
```

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

In [11]:
```
planets[3:]
```

Out[11]:
```
['Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

i.e. the expression above means "give me all the planets from index 3 onward".

We can also use negative indices when slicing:

In [12]:
```
# All the planets except the first and last
planets[1:-1]
```

Out[12]:
```
['Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus']
```

In [13]:
```
# The last 3 planets
planets[-3:]
```

Out[13]:
```
['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:

In [14]:
```
planets[3] = 'Malacandra'
planets
```

Out[14]:
```
['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.

In [15]:
```
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:

In [16]:
```
# How many planets are there?
len(planets)
```

Out[16]:
```
8
```

`sorted` returns a sorted version of a list:

In [17]:
```
# The planets sorted in alphabetical order
sorted(planets)
```

Out[17]:
```
['Earth', 'Jupiter', 'Mars', 'Mercury', 'Neptune', 'Saturn', 'Uranus', 'Ven
us']
```

`sum` does what you might expect:

In [18]:
```
primes = [2, 3, 5, 7]
sum(primes)
```

Out[18]: 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.

In [19]:
```
max(primes)
```

Out[19]: 7

Interlude: objects

I've 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.)

In [20]:
```
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:

In [21]:
```
x.bit_length
```

Out[21]: <function int.bit_length()>

To actually call it, we add parentheses:

In [22]:
```
x.bit_length()
```

Out[22]: 4

> **Aside:** You've actually been calling methods already if you've been doing the exercises. In the exercise notebooks `q1`, `q2`, `q3`, etc. are all objects which have methods called `check`, `hint`, and `solution`.

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

In [23]:
```
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:

In [24]:
```
# 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)`.

> **Aside:** `append` is a method carried around by *all* objects of type list, not just `planets`, so we also could have called `help(list.append)`. However, if we try to call `help(append)`, Python will complain that no variable exists called "append". The "append" name only exists within lists - it doesn't exist as a standalone name like builtin functions such as `max` or `len`.

In [25]:
```
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`:

In [26]:
```
planets
```

Out[26]:
```
['Mercury',
 'Venus',
 'Earth',
 'Mars',
 'Jupiter',
 'Saturn',
 'Uranus',
 'Neptune',
 'Pluto']
```

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

In [27]:
```
planets.pop()
```

Out[27]:
```
'Pluto'
```

In [28]:
```
planets
```

Out[28]:
```
['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.

In [29]:
```
planets.index('Earth')
```

Out[29]:
```
2
```

It comes third (i.e. at index 2 - 0 indexing!).

At what index does Pluto occur?

In [30]:
```
planets.index('Pluto')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[30], line 1
----> 1 planets.index('Pluto')

ValueError: 'Pluto' is not in list
```

Oh, that's right...

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

In [31]:
```
# Is Earth a planet?
"Earth" in planets
```

Out[31]:
```
True
```

In [32]:
```
# Is Calbefraques a planet?
"Calbefraques" in planets
```

Out[32]:
```
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:

In [33]:
```
help(planets)
```

```
Help on list object:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iadd__(self, value, /)
 |      Implement self+=value.
 |
 |  __imul__(self, value, /)
 |      Implement self*=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
```

```
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __reversed__(self, /)
 |      Return a reverse iterator over the list.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __setitem__(self, key, value, /)
 |      Set self[key] to value.
 |
 |  __sizeof__(self, /)
 |      Return the size of the list in memory, in bytes.
 |
 |  append(self, object, /)
 |      Append object to the end of the list.
 |
 |  clear(self, /)
 |      Remove all items from list.
 |
 |  copy(self, /)
 |      Return a shallow copy of the list.
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  extend(self, iterable, /)
 |      Extend list by appending elements from the iterable.
 |
 |  index(self, value, start=0, stop=9223372036854775807, /)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  insert(self, index, object, /)
 |      Insert object before index.
 |
 |  pop(self, index=-1, /)
 |      Remove and return item at index (default last).
 |
 |      Raises IndexError if list is empty or index is out of range.
 |
 |  remove(self, value, /)
 |      Remove first occurrence of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  reverse(self, /)
```

```
 |      Reverse *IN PLACE*.
 |
 |  sort(self, /, *, key=None, reverse=False)
 |      Sort the list in ascending order and return None.
 |
 |      The sort is in-place (i.e. the list itself is modified) and stable
(i.e. the
 |      order of two equal elements is maintained).
 |
 |      If a key function is given, apply it once to each list item and sort
them,
 |      ascending or descending, according to their function values.
 |
 |      The reverse flag can be set to sort in descending order.
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signatu
re.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

Click the "output" button to see the full help page. Lists have lots of methods with weird-looking names like `__eq__` and `__iadd__`. Don't worry too much about these for now. (You'll probably never call such methods directly. But they get called behind the scenes when we use syntax like indexing or comparison operators.) The most interesting methods are toward the bottom of the list (`append`, `clear`, `copy`, etc.).

Tuples

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

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

In [34]:
```
t = (1, 2, 3)
```

In [35]:
```
t = 1, 2, 3 # equivalent to above
t
```

2: They cannot be modified (they are *immutable*).

In [36]:
```
t[0] = 100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[36], line 1
----> 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:

In [37]:
```
x = 0.125
x.as_integer_ratio()
```

Out[37]: (1, 8)

These multiple return values can be individually assigned as follows:

In [38]:
```
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!

In [39]:
```
a = 1
b = 0
a, b = b, a
print(a, b)
```

```
0 1
```

Your Turn

You learn best by writing code, not just reading it. So try **the coding challenge** now.
