This lesson will cover two essential Python types: **strings** and **dictionaries**.

Strings

One place where the Python language really shines is in the manipulation of strings. This section will cover some of Python's built-in string methods and formatting operations.

Such string manipulation patterns come up often in the context of data science work.

String syntax

You've already seen plenty of strings in examples during the previous lessons, but just to recap, strings in Python can be defined using either single or double quotations. They are functionally equivalent.

```
In [1]: x = 'Pluto is a planet'
        y = "Pluto is a planet"
        x == y
```

```
Out[1]: True
```

Double quotes are convenient if your string contains a single quote character (e.g. representing an apostrophe).

Similarly, it's easy to create a string that contains double-quotes if you wrap it in single quotes:

```
In [2]: print("Pluto's a planet!")
        print('My dog is named "Pluto"')
```

```
Pluto's a planet!
My dog is named "Pluto"
```

If we try to put a single quote character inside a single-quoted string, Python gets confused:

```
In [3]: 'Pluto's a planet!'
```

```
  Cell In[3], line 1
    'Pluto's a planet!'
                      ^
SyntaxError: unterminated string literal (detected at line 1)
```

We can fix this by "escaping" the single quote with a backslash.

```
In [4]: 'Pluto\'s a planet!'
```

Loading [MathJax]/extensions/Safe.js

Out[4]:

```
"Pluto's a planet!"
```

The table below summarizes some important uses of the backslash character.

| What you type... | What you get | example | `print(example)` | |--------------|-------
---------|---|| `\'` | `'` |
`'What\'s up?'` | `What's up?` |
| `\"` | `"` | `"That's \"cool\""` | `That's "cool"` |
| `\\` | `\` | `"Look, a mountain: /\\"` | `Look, a mountain: /\` || `\n` |
| `"1\n2 3"` | `1`
`2 3` |

The last sequence, `\n`, represents the *newline character*. It causes Python to start a new line.

In [5]:

```
hello = "hello\nworld"
print(hello)
```

```
hello
world
```

In addition, Python's triple quote syntax for strings lets us include newlines literally (i.e. by just hitting 'Enter' on our keyboard, rather than using the special '\n' sequence). We've already seen this in the docstrings we use to document our functions, but we can use them anywhere we want to define a string.

In [6]:

```
triplequoted_hello = """hello
world"""
print(triplequoted_hello)
triplequoted_hello == hello
```

```
hello
world
```

Out[6]:

```
True
```

The `print()` function automatically adds a newline character unless we specify a value for the keyword argument `end` other than the default value of `'\n'`:

In [7]:

```
print("hello")
print("world")
print("hello", end='')
print("pluto", end='')
```

```
hello
world
hellopluto
```

Strings are sequences

Strings can be thought of as sequences of characters. Almost everything we've seen that we can do to a list, we can also do to a string.

In [8]:
```
# Indexing
planet = 'Pluto'
planet[0]
```

Out[8]:
```
'P'
```

In [9]:
```
# Slicing
planet[-3:]
```

Out[9]:
```
'uto'
```

In [10]:
```
# How long is this string?
len(planet)
```

Out[10]:
```
5
```

In [11]:
```
# Yes, we can even loop over them
[char+'! ' for char in planet]
```

Out[11]:
```
['P! ', 'l! ', 'u! ', 't! ', 'o! ']
```

But a major way in which they differ from lists is that they are *immutable*. We can't modify them.

In [12]:
```
planet[0] = 'B'
# planet.append doesn't work either
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[12], line 1
----> 1 planet[0] = 'B'
      2 # planet.append doesn't work either

TypeError: 'str' object does not support item assignment
```

String methods

Like `list`, the type `str` has lots of very useful methods. I'll show just a few examples here.

In [13]:
```
# ALL CAPS
claim = "Pluto is a planet!"
claim.upper()
```

Out[13]:
```
'PLUTO IS A PLANET!'
```

In [14]:
```
# all lowercase
claim.lower()
```

Loading [MathJax]/extensions/Safe.js

```
Out[14]: 'pluto is a planet!'
```

```
In [15]: # Searching for the first index of a substring
         claim.index('plan')
```

```
Out[15]: 11
```

```
In [16]: claim.startswith(planet)
```

```
Out[16]: True
```

```
In [17]: # false because of missing exclamation mark
         claim.endswith('planet')
```

```
Out[17]: False
```

Going between strings and lists: `.split()` and `.join()`

`str.split()` turns a string into a list of smaller strings, breaking on whitespace by default. This is super useful for taking you from one big string to a list of words.

```
In [18]: words = claim.split()
         words
```

```
Out[18]: ['Pluto', 'is', 'a', 'planet!']
```

Occasionally you'll want to split on something other than whitespace:

```
In [19]: datestr = '1956-01-31'
         year, month, day = datestr.split('-')
```

`str.join()` takes us in the other direction, sewing a list of strings up into one long string, using the string it was called on as a separator.

```
In [20]: '/'.join([month, day, year])
```

```
Out[20]: '01/31/1956'
```

```
In [21]: # Yes, we can put unicode characters right in our string literals :)
         ' 👏 '.join([word.upper() for word in words])
```

```
Out[21]: 'PLUTO 👏 IS 👏 A 👏 PLANET!'
```

Building strings with `.format()`

Python lets us concatenate strings with the `+` operator.

```
In [22]: planet + ', we miss you.'
```

Loading [MathJax]/extensions/Safe.js

```
Out[22]: 'Pluto, we miss you.'
```

If we want to throw in any non-string objects, we have to be careful to call `str()` on them first

```
In [23]: position = 9
         planet + ", you'll always be the " + position + "th planet to me."
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[23], line 2
      1 position = 9
----> 2 planet + ", you'll always be the " + position + "th planet to me."

TypeError: can only concatenate str (not "int") to str
```

```
In [24]: planet + ", you'll always be the " + str(position) + "th planet to me."
```

```
Out[24]: "Pluto, you'll always be the 9th planet to me."
```

This is getting hard to read and annoying to type. `str.format()` to the rescue.

```
In [25]: "{}, you'll always be the {}th planet to me.".format(planet, position)
```

```
Out[25]: "Pluto, you'll always be the 9th planet to me."
```

So much cleaner! We call `.format()` on a "format string", where the Python values we want to insert are represented with `{}` placeholders.

Notice how we didn't even have to call `str()` to convert `position` from an int. `format()` takes care of that for us.

If that was all that `format()` did, it would still be incredibly useful. But as it turns out, it can do a *lot* more. Here's just a taste:

```
In [26]: pluto_mass = 1.303 * 10**22
         earth_mass = 5.9722 * 10**24
         population = 52910390
         #         2 decimal points   3 decimal points, format as percent     separat
         "{} weighs about {:.2} kilograms ({:.3%} of Earth's mass). It is home to {:,
             planet, pluto_mass, pluto_mass / earth_mass, population,
         )
```

```
Out[26]: "Pluto weighs about 1.3e+22 kilograms (0.218% of Earth's mass). It is home
         to 52,910,390 Plutonians."
```

```
In [27]: # Referring to format() arguments by index, starting from 0
         s = """Pluto's a {0}.
         No, it's a {1}.
         {0}!
         {1}!""".format('planet', 'dwarf planet')
```

Loading [MathJax]/extensions/Safe.js

```
Pluto's a planet.
No, it's a dwarf planet.
planet!
dwarf planet!
```

You could probably write a short book just on `str.format`, so I'll stop here, and point you to pyformat.info and the official docs for further reading.

Building strings with f-string

This is an alternative modernized way of building strings came with Python 3.6+

In [28]:

```
f"{planet}, you'll always be the {position}th planet to me."
```

Out[28]:

```
"Pluto, you'll always be the 9th planet to me."
```

Comparing to `.format()` command, f-string is cleaner. Especially, to do something like this

In [29]:

```
pluto_mass = 1.303 * 10**22
earth_mass = 5.9722 * 10**24
population = 52910390
pluto_earth_ratio = pluto_mass / earth_mass

f"{planet} weighs about {pluto_mass:.2} kilograms ({pluto_earth_ratio:.3%} c
```

Out[29]:

```
"Pluto weighs about 1.3e+22 kilograms (0.218% of Earth's mass). It is home
to 52,910,390 Plutonians."
```

Dictionaries

Dictionaries are a built-in Python data structure for mapping keys to values.

In [30]:

```
numbers = {'one':1, 'two':2, 'three':3}
```

In this case `'one'`, `'two'`, and `'three'` are the **keys**, and 1, 2 and 3 are their corresponding values.

Values are accessed via square bracket syntax similar to indexing into lists and strings.

In [31]:

```
numbers['one']
```

Out[31]:

```
1
```

We can use the same syntax to add another key, value pair

In [32]:

```
numbers['eleven'] = 11
numbers
```

Loading [MathJax]/extensions/Safe.js

```
Out[32]: {'one': 1, 'two': 2, 'three': 3, 'eleven': 11}
```

Or to change the value associated with an existing key

```
In [33]: numbers['one'] = 'Pluto'
         numbers
```

```
Out[33]: {'one': 'Pluto', 'two': 2, 'three': 3, 'eleven': 11}
```

Python has *dictionary comprehensions* with a syntax similar to the list comprehensions we saw in the previous tutorial.

```
In [34]: planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus
         planet_to_initial = {planet: planet[0] for planet in planets}
         planet_to_initial
```

```
Out[34]: {'Mercury': 'M',
          'Venus': 'V',
          'Earth': 'E',
          'Mars': 'M',
          'Jupiter': 'J',
          'Saturn': 'S',
          'Uranus': 'U',
          'Neptune': 'N'}
```

The `in` operator tells us whether something is a key in the dictionary

```
In [35]: 'Saturn' in planet_to_initial
```

```
Out[35]: True
```

```
In [36]: 'Betelgeuse' in planet_to_initial
```

```
Out[36]: False
```

A for loop over a dictionary will loop over its keys

```
In [37]: for k in numbers:
             print("{} = {}".format(k, numbers[k]))
```

```
one = Pluto
two = 2
three = 3
eleven = 11
```

We can access a collection of all the keys or all the values with `dict.keys()` and `dict.values()`, respectively.

```
In [38]: # Get all the initials, sort them alphabetically, and put them in a space-se
         ' '.join(sorted(planet_to_initial.values()))
```

```
Loading [MathJax]/extensions/Safe.js S U V'
```

The very useful `dict.items()` method lets us iterate over the keys and values of a dictionary simultaneously. (In Python jargon, an **item** refers to a key, value pair)

In [39]:
```
for planet, initial in planet_to_initial.items():
    print("{} begins with \"{}\"".format(planet.rjust(10), initial))
```

```
   Mercury begins with "M"
     Venus begins with "V"
     Earth begins with "E"
      Mars begins with "M"
   Jupiter begins with "J"
    Saturn begins with "S"
    Uranus begins with "U"
   Neptune begins with "N"
```

To read a full inventory of dictionaries' methods, click the "output" button below to read the full help page, or check out the official online documentation.

In [40]:
```
help(dict)
```

```
Help on class dict in module builtins:

class dict(object)
 |  dict() -> new empty dictionary
 |  dict(mapping) -> new dictionary initialized from a mapping object's
 |      (key, value) pairs
 |  dict(iterable) -> new dictionary initialized as if via:
 |      d = {}
 |      for k, v in iterable:
 |          d[k] = v
 |  dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |      in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |  Built-in subclasses:
 |      StgDict
 |
 |  Methods defined here:
 |
 |  __contains__(self, key, /)
 |      True if the dictionary has the specified key, else False.
 |
 |  __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.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __ior__(self, value, /)
 |      Return self|=value.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
```

```
  |
  |  __ne__(self, value, /)
  |      Return self!=value.
  |
  |  __or__(self, value, /)
  |      Return self|value.
  |
  |  __repr__(self, /)
  |      Return repr(self).
  |
  |  __reversed__(self, /)
  |      Return a reverse iterator over the dict keys.
  |
  |  __ror__(self, value, /)
  |      Return value|self.
  |
  |  __setitem__(self, key, value, /)
  |      Set self[key] to value.
  |
  |  __sizeof__(...)
  |      D.__sizeof__() -> size of D in memory, in bytes
  |
  |  clear(...)
  |      D.clear() -> None.  Remove all items from D.
  |
  |  copy(...)
  |      D.copy() -> a shallow copy of D
  |
  |  get(self, key, default=None, /)
  |      Return the value for key if key is in the dictionary, else default.
  |
  |  items(...)
  |      D.items() -> a set-like object providing a view on D's items
  |
  |  keys(...)
  |      D.keys() -> a set-like object providing a view on D's keys
  |
  |  pop(...)
  |      D.pop(k[,d]) -> v, remove specified key and return the corresponding
value.
  |
  |      If the key is not found, return the default if given; otherwise,
  |      raise a KeyError.
  |
  |  popitem(self, /)
  |      Remove and return a (key, value) pair as a 2-tuple.
  |
  |      Pairs are returned in LIFO (last-in, first-out) order.
  |      Raises KeyError if the dict is empty.
  |
  |  setdefault(self, key, default=None, /)
  |      Insert key with a value of default if key is not in the dictionary.
  |
  |      Return the value for key if key is in the dictionary, else default.
  |
  |  update(...)
```

```
 |      D.update([E, ]**F) -> None.  Update D from dict/iterable E and F.
 |      If E is present and has a .keys() method, then does:  for k in E: D
[k] = E[k]
 |      If E is present and lacks a .keys() method, then does:  for k, v in
E: D[k] = v
 |      In either case, this is followed by: for k in F:  D[k] = F[k]
 |
 |  values(...)
 |      D.values() -> an object providing a view on D's values
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585
 |
 |  fromkeys(iterable, value=None, /) from builtins.type
 |      Create a new dictionary with keys from iterable and values set to va
lue.
 |
 |  ----------------------------------------------------------------------
 |  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
```

Your Turn

You've learned a lot of Python... go **demonstrate your new skills** with some realistic programming applications.