Python Data Model

June 4, 2020

1 Resources

- Official documentation link
- On Understanding Types, Data Abstraction, and Polymorphism article
- Washington University Data Programming course slides
- A tour of Python slides

2 Python's Data Model

- Everything is an object
- Each object has an identity, a type, and a value
- id(obj) returns the object's identity
- type(obj) returns the object's type

2.1 Identity

The identity of an object can never change

– (Currently) implemented as the object's address in memory.

– You can check to see if two objects are identical with the keyword is

2.2 Type

- The type of an object cannot change
- It specifies two things:

– what operatons are allowed

– the set of values the object can hold

- An object's identity can never change
- An object's type can never change

- An object's value can never change, unless it has a mutable type

3 Types: some definitions and context

Some historical languages were untyped

- You could, say, divide a string by a number, and the program would continue.
- The result was still nonsense, of course, and program behavior was completely undefined.
- This was considered unacceptable

Modern languages may be staticly typed or dynamically typed

- "staticly typed" means that types are assigned before the program is executed
- "dynamically typed" means that types are assigned (and type errors caught) at runtime

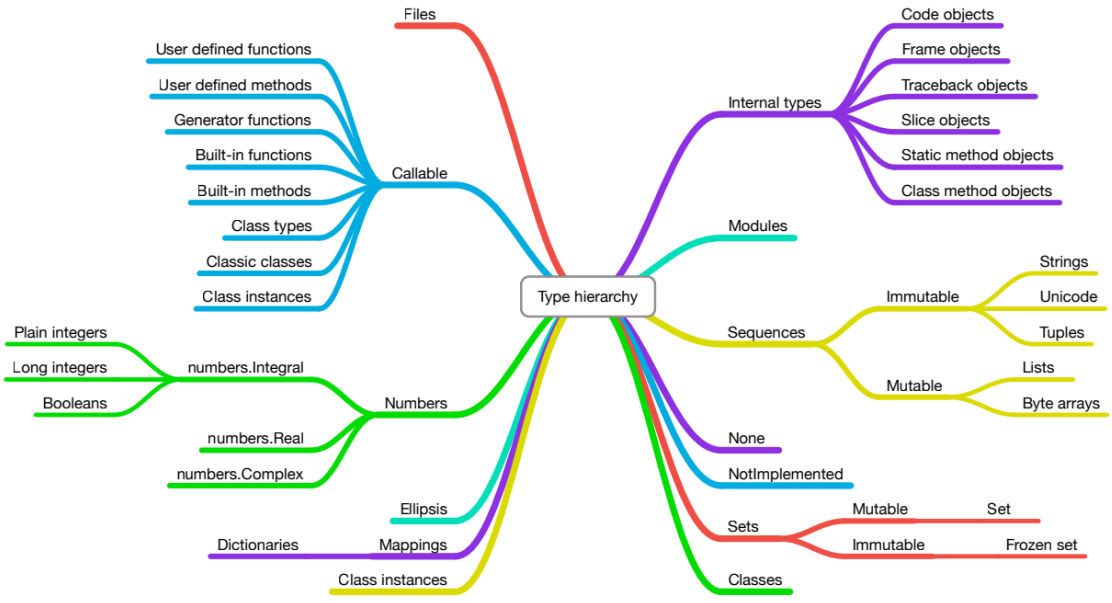
Modern languages may be strongly typed or weakly typed

- For our purposes, "weakly typed" means the language supports a significant number of implicit type conversions.

For example, (5 + "3") could trigger a conversion from "3" to 3

For our purposes, Python can be considered

- strongly typed
- dynamically typed



3.1 Guess the Types

[1]:
```
def mbar_to_mmHg(pressure):
    return pressure * 0.75006
```

[2]:
```
def abs(x):
    if val < 0:
        return -1 * val
    else:
        return 1 * val
```

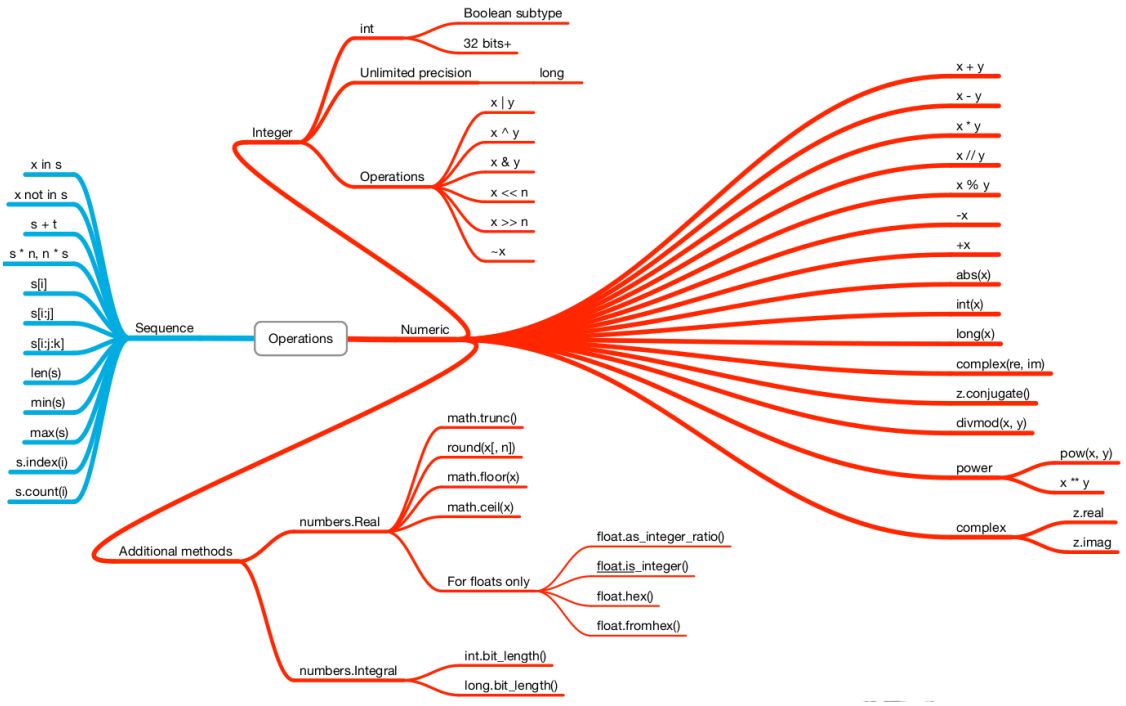
[3]:
```
def debug(x):
    print(x)
```

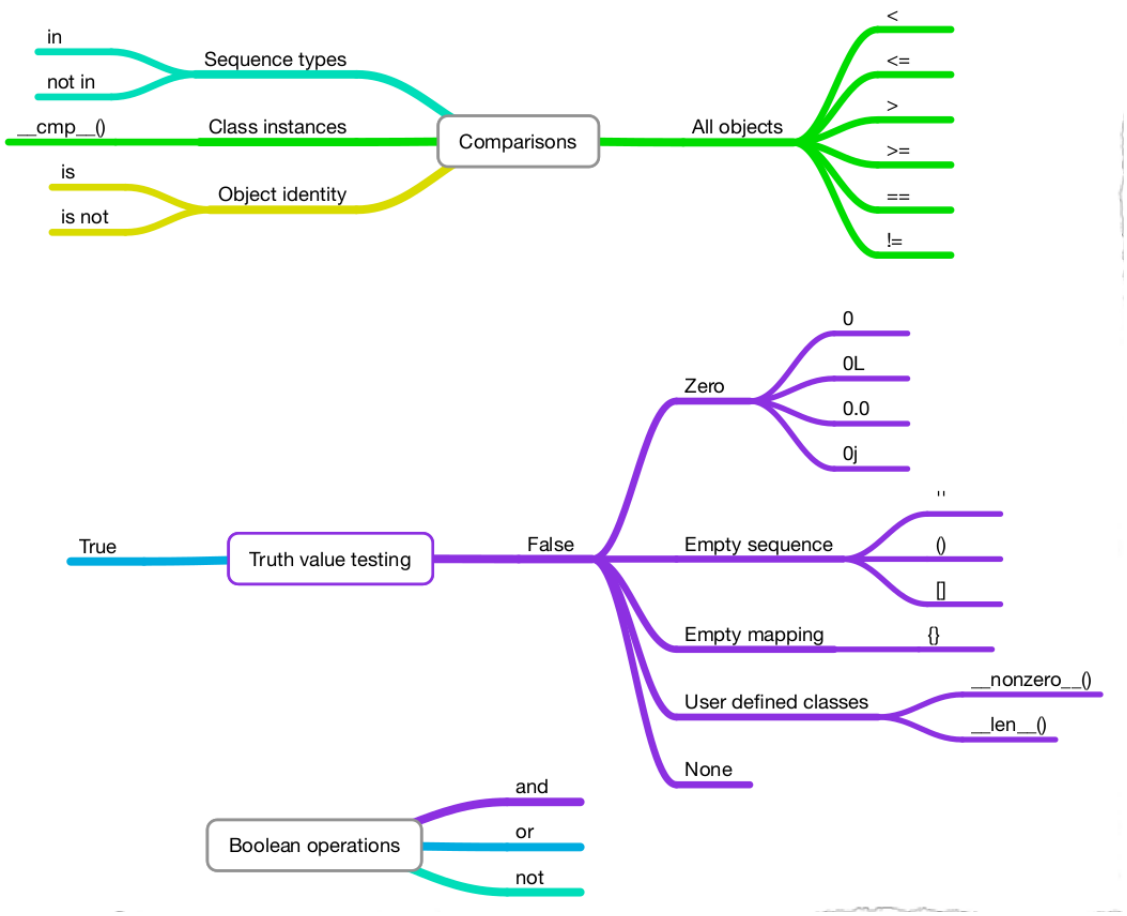
3.2 Duck Typing

"If it walks like a duck and it talks like a duck, then it must be a duck."

(Note: this analogy can be misleading!)

At runtime, the operands are checked to make sure they support the requested operation.





3.3 Explore basic data types

Let's try some simple Python commands to explore numbers, strings, and lists.

3.3.1 Numbers

The Python interpreter can act as a simple calculator: type an expression at it outputs the value.

Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` work just like in most other programming languages (such as Pascal or C); parentheses (`()`) can be used for grouping and order or precedence. For example:

[4]:
```
2 + 2
```

[4]: 4

[5]:
```
50 - 5*6
```

[5]: 20

[6]:
```
(50 - 5*6) / 4
```

[6]:
```
5.0
```

[7]:
```
8 / 5  # Division always returns a floating point number.
```

[7]:
```
1.6
```

The integer numbers (e.g. `2`, `4`, `20`) have type `int`, the ones with a fractional part (e.g. `5.0`, `1.6`) have type `float`. We'll see more about numeric types later in the tutorial.

Division (`/`) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the `//` operator; to calculate the remainder you can use `%`:

[8]:
```
17 / 3  # Classic division returns a float.
```

[8]:
```
5.666666666666667
```

[9]:
```
17 // 3  # Floor division discards the fractional part.
```

[9]:
```
5
```

[10]:
```
17 % 3  # The % operator returns the remainder of the division.
```

[10]:
```
2
```

[11]:
```
5 * 3 + 2  # result * divisor + remainder
```

[11]:
```
17
```

Use the `**` operator to calculate powers:

[12]:
```
5 ** 2  # 5 squared
```

[12]:
```
25
```

[13]:
```
2 ** 7  # 2 to the power of 7
```

[13]:
```
128
```

`**` has higher precedence than `-`; if you want a negative base, use parentheses:

[14]:
```
-3**2  # Same as -(3**2)
```

[14]:
```
-9
```

[15]:
```
(-3)**2
```

[15]:
```
9
```

The equal sign (=) assigns a value to a variable:

[16]:
```
width = 20
height = 5 * 90
width * height
```

[16]:
```
9000
```

If a variable is not "defined" (assigned a value), using it produces an error:

[17]:
```
n  # Try to access an undefined variable.
```

```
␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-17-2d383632fd8e> in <module>
   ----> 1 n  # Try to access an undefined variable.


       NameError: name 'n' is not defined
```

Python provides full support for floating point numbers; operators with mixed type operands convert the integer operand to floating point:

[19]:
```
3 * 3.75 / 1.5
```

[19]:
```
7.5
```

[20]:
```
7.0 / 2
```

[20]:
```
3.5
```

In interactive mode and in Jupyter notebooks, the last printed expression is assigned to the variable _. This means that when you're using Python as a desk calculator, its somewhat easier to continue calculations. For example:

[21]:
```
tax = 12.5 / 100
price = 100.50
price * tax
```

[21]:
```
12.5625
```

[22]:
```
price + _
```

[22]:
```
113.0625
```

[23]:
```
round(_, 2)
```

[23]:
```
113.06
```

Treat the `_` variable as read-only. Don't explicitly assign a value to it–doing so creates an independent local variable with the same name, thereby masking the built-in variable with its magic behavior.

In addition to `int` and `float`, Python supports other types of numbers, such as `Decimal` and `Fraction`. Python also has built-in support for complex numbers, and uses the `j` or `J` suffix to indicate the imaginary part (e.g. `3+5j`).

3.3.2 Strings

Besides numbers, Python can also manipulate strings. Strings can enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result. Use `\` to escape quotes, that is, to use a quote within the string itself:

[24]:
```
'spam eggs'  # Single quotes.
```

[24]:
```
'spam eggs'
```

[25]:
```
'doesn\'t'  # Use \' to escape the single quote...
```

[25]:
```
"doesn't"
```

[26]:
```
"doesn't"  # ...or use double quotes instead.
```

[26]:
```
"doesn't"
```

[27]:
```
'"Yes," he said.'
```

[27]:
```
'"Yes," he said.'
```

[28]:
```
"\"Yes,\" he said."
```

[28]:
```
'"Yes," he said.'
```

[29]:
```
'"Isn\'t," she said.'
```

[29]:
```
'"Isn\'t," she said.'
```

In the interactive interpreter and Jupyter notebooks, the output string is enclosed in quotes and special characters are escaped with backslashes. Although this output sometimes looks different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise its

enclosed in single quotes. The `print()` function produces a more readable output by omitting the enclosing quotes and by printing escaped and special characters:

[30]:
```
'"Isn\'t," she said.'
```

[30]:
```
'"Isn\'t," she said.'
```

[31]:
```
print('"Isn\'t," she said.')
```

```
"Isn't," she said.
```

[32]:
```
s = 'First line.\nSecond line.'  # \n means newline.
s  # Without print(), \n is included in the output.
```

[32]:
```
'First line.\nSecond line.'
```

[33]:
```
print(s)  # With print(), \n produces a new line.
```

```
First line.
Second line.
```

If you don't want escaped characters (prefaced by \) to be interpreted as special characters, use *raw strings* by adding an `r` before the first quote:

[34]:
```
print('C:\some\name')  # Here \n means newline!
```

```
C:\some
ame
```

[35]:
```
print(r'C:\some\name')  # Note the r before the quote.
```

```
C:\some\name
```

String literals can span multiple lines and are delineated by triple-quotes: `"""..."""` or `'''...'''`. End of lines are automatically included in the string, but it's possible to prevent this by adding a \ at the end of the line. For example, without a \, the following example includes an extra line at the beginning of the output:

[36]:
```
print("""
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Adding a \ removes that extra line:

[37]:
```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Because Python doesn't provide a means for creating multi-line comments, developers often just use triple quotes for this purpose. In a Jupyter notebook, however, such quotes define a string literal which appears as the output of a code cell:

[38]:
```
"""
Everything between the first three quotes, including new lines,
is part of the multi-line comment. Technically, the Python interpreter
simply sees the comment as a string, and because it's not otherwise
used in code, the string is ignored. Convenient, eh?
"""
```

[38]:
```
"\nEverything between the first three quotes, including new lines,\nis part of
the multi-line comment. Technically, the Python interpreter\nsimply sees the
comment as a string, and because it's not otherwise\nused in code, the string is
ignored. Convenient, eh?\n"
```

For this reason, it's best in notebooks to use the # comment character at the beginning of each line, or better still, just use a Markdown cell!

Strings can be *concatenated* (glued together) with the + operator, and repeated with *:

[39]:
```
# 3 times 'un', followed by 'ium'
3 * 'un' + 'ium'
```

[39]:
```
'unununium'
```

Two or more *string literals* (that is, the values enclosed in quotes) placed next to each other are automatically concatenated:

[40]:
```
'Py' 'thon'
```

[40]:
```
'Python'
```

Automatic concatenation works only with two literals; it does not work with variables or expressions, so the following cell produces an error:

[41]:
```
prefix = 'Py'
prefix 'thon'  # Can't concatenate a variable and a string literal.
```

```
  File "<ipython-input-41-00ad70cd97bc>", line 2
    prefix 'thon'  # Can't concatenate a variable and a string literal.
                ^
SyntaxError: invalid syntax
```

The following cell likewise produces an error:

[43]:
```
('un' * 3) 'ium'
```

```
  File "<ipython-input-43-f4764cbe42a8>", line 1
    ('un' * 3) 'ium'
                  ^
SyntaxError: invalid syntax
```

To concatenate variables, or a variable and a literal, use `+`:

[44]:
```
prefix = 'Py'
prefix + 'thon'
```

[44]:
```
'Python'
```

Automatic concatenation is particularly useful when you want to break up long strings:

[45]:
```
text = ('Put several strings within parentheses '
            'to have them joined together.')
text
```

[45]:
```
'Put several strings within parentheses to have them joined together.'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

[46]:
```
word = 'Python'
word[0]  # Character in position 0.
```

[46]:
```
'P'
```

[47]:
```
word[5]  # Character in position 5.
```

[47]:
```
'n'
```

Indices may also be negative numbers, which means to start counting from the end of the string. Note that because -0 is the same as 0, negative indices start from -1:

[48]:
```
word[-1]  # Last character.
```

[48]:
```
'n'
```

[49]:
```
word[-2]  # Second-last character.
```

[49]:
```
'o'
```

[50]:
```
word[-6]
```

[50]:
```
'P'
```

In addition to indexing, which extracts individual characters, Python also supports *slicing*, which extracts a substring. To slide, you indicate a *range* in the format `start:end`, where the start position is included but the end position is excluded:

[51]:
```
word[0:2]  # Characters from position 0 (included) to 2 (excluded).
```

[51]:
```
'Py'
```

[52]:
```
word[2:5]  # Characters from position 2 (included) to 5 (excluded).
```

[52]:
```
'tho'
```

If you omit either position, the default start position is 0 and the default end is the length of the string:

[53]:
```
word[:2]   # Character from the beginning to position 2 (excluded).
```

[53]:
```
'Py'
```

[54]:
```
word[4:]  # Characters from position 4 (included) to the end.
```

[54]:
```
'on'
```

[55]:
```
word[-2:] # Characters from the second-last (included) to the end.
```

[55]:
```
'on'
```

This characteristic means that `s[:i] + s[i:]` is always equal to `s`:

[56]:
```
word[:2] + word[2:]
```

[56]:
```
'Python'
```

[57]:
```
word[:4] + word[4:]
```

[57]: 'Python'

One way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of n characters has index n. For example: +—+—+—+—+—+—+ | P | y | t | h | o | n | +—+—+—+—+—+—+ 0 1 2 3 4 5 6 -6 -5 -4 -3 -2 -1 The first row of numbers gives the position of the indices 0…6 in the string; the second row gives the corresponding negative indices. The slice from i to j consists of all characters between the edges labeled i and j, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large results in an error:

[58]:
```
word[42]  # The word only has 6 characters.
```

```
␣
↪---------------------------------------------------------------------------

IndexError                                Traceback (most recent call␣
↪last)

<ipython-input-58-e894f93573ea> in <module>
----> 1 word[42]  # The word only has 6 characters.


IndexError: string index out of range
```

However, when used in a range, an index that's too large defaults to the size of the string and does not give an error. This characteristic is useful when you always want to slice at a particular index regardless of the length of a string:

[59]:
```
word[4:42]
```

[59]: 'on'

[60]:
```
word[42:]
```

[60]: ''

Python strings are immutable, which means they cannot be changed. Therefore, assigning a value to an indexed position in a string results in an error:

[61]:
```
word[0] = 'J'
```

```
␣
↪---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-61-91a956888ca7> in <module>
----> 1 word[0] = 'J'


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

The following cell also produces an error:

[63]:
```
word[2:] = 'py'
```

```
␣
↪--------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-63-6488bbf78f5a> in <module>
----> 1 word[2:] = 'py'


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

A slice it itself a value that you can concatenate with other values using +:

[ ]:
```
'J' + word[1:]
```

[ ]:
```
word[:2] + 'Py'
```

A slice, however, is not a string literal and cannot be used with automatic concatenation. The following code produces an error:

[64]:
```
word[:2] 'Py'    # Slice is not a literal; produces an error
```

```
  File "<ipython-input-64-60be1c701626>", line 1
    word[:2] 'Py'    # Slice is not a literal; produces an error
               ^
SyntaxError: invalid syntax
```

The built-in function `len()` returns the length of a string:

[65]:
```
s = 'supercalifragilisticexpialidocious'
len(s)
```

[65]:
```
34
```

3.3.3 Lists

Python knows a number of *compound* data types, which are used to group together other values. The most versatile is the *list*, which can be written as a sequence of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

[66]:
```
squares = [1, 4, 9, 16, 25]
squares
```

[66]:
```
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in sequence types), lists can be indexed and sliced:

[67]:
```
squares[0]  # Indexing returns the item.
```

[67]:
```
1
```

[68]:
```
squares[-1]
```

[68]:
```
25
```

[69]:
```
squares[-3:]  # Slicing returns a new list.
```

[69]:
```
[9, 16, 25]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a new (shallow) copy of the list:

[70]:
```
squares[:]
```

[70]:
```
[1, 4, 9, 16, 25]
```

Lists also support concatenation with the + operator:

[71]:
```
squares + [36, 49, 64, 81, 100]
```

[71]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, which means you can change any value in the list:

[72]:
```
cubes = [1, 8, 27, 65, 125]  # Something's wrong here ...
4 ** 3  # the cube of 4 is 64, not 65!
```

[72]:
```
64
```

[73]:
```
cubes[3] = 64  # Replace the wrong value.
cubes
```

[73]:
```
[1, 8, 27, 64, 125]
```

Use the list's `append()` method to add new items to the end of the list:

[74]:
```
cubes.append(216)  # Add the cube of 6 ...
cubes.append(7 ** 3)  # and the cube of 7.
cubes
```

[74]:
```
[1, 8, 27, 64, 125, 216, 343]
```

You can even assign to slices, which can change the size of the list or clear it entirely:

[75]:
```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
letters
```

[75]:
```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

[76]:
```
# Replace some values.
letters[2:5] = ['C', 'D', 'E']
letters
```

[76]:
```
['a', 'b', 'C', 'D', 'E', 'f', 'g']
```

[77]:
```
# Now remove them.
letters[2:5] = []
letters
```

[77]:
```
['a', 'b', 'f', 'g']
```

[78]:
```
# Clear the list by replacing all the elements with an empty list.
letters[:] = []
letters
```

[78]:
```
[]
```

The built-in `len()` function also applies to lists:

[79]:
```
letters = ['a', 'b', 'c', 'd']
len(letters)
```

[79]:
```
4
```

You can nest lists, which means to create lists that contain other lists. For example:

[80]:
```
a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n]
x
```

[80]:
```
[['a', 'b', 'c'], [1, 2, 3]]
```

[81]:
```
x[0]
```

[81]:
```
['a', 'b', 'c']
```

[82]:
```
x[0][1]
```

[82]:
```
'b'
```