CS 42—Python sequences

Tuesday, October 30, 2018

Summary

Today, we'll learn about **sequences** in Python. Sequences include lists, tuples, and strings. We'll practice how to **slice** a sequence (which creates a new sequence that contains specific elements of the original sequence). We'll also practice **list comprehensions**, a convenient way to create lists. Along the way, we'll also revisit an old friend: functional programming.

First, though, we start with a review of namespaces and talk about how libraries (i.e., **modules**) are just fancy namespaces.

Modules: they're just more namespaces!

Resolving a module's name

When Python sees a statement like this:

```
import numbers
```

Python looks for a file called `numbers.py` using the following procedure:

1. Look for the file in the **current directory**. If found, proceed with the `import`.
2. If there is no such file in the current directory, look in the **module search path**.[1] If found, proceed with the `import`.
3. If the file isn't in the module search path, throw an `ImportError`.

An `import` statement binds a name to a namespace

When Python sees the line:

```
import numbers
```

it performs the following steps:

1. Create a new namespace (which we'll refer to as *N*).
2. Run `numbers.py` as if it were a program, using *N* as the global (and currently active) namespace.
3. Bind the name `numbers` to *N* in the originally active namespace.

[1] The module search path is a list of directories that includes the standard-library directories. We almost certainly won't need to configure this list in CS 42; but if you're interested, there's more information here: bit.ly/2e7LyBe.

Other forms of `import` also affect bindings

Python provides several forms of the `import` statement, each of which gives you a certain amount of control over how names from the module are bound.

`from … import …`

When Python sees the line:

```
from numbers import double
```

it does steps 1 and 2 from above, then it does the following:

3. Copy the binding for `double` to the originally active namespace.

When Python sees the line:

```
from numbers import double, triple
```

it does steps 1 and 2 from above, then it does the following:

3. Copy the bindings for `double` and `triple` to the originally active namespace.

`from … import *`

When Python sees the line:

```
from numbers import *
```

it does steps 1 and 2 from above, then it does the following:

3. Copy *all* the bindings from the module to the originally active namespace.

Good programming practice: Using **`from … import *`** is usually considered bad style—we should specifically `import` the bindings we need.

`import … as …`

When Python sees the line:

```
import numbers as num
```

it does steps 1 and 2 from above, then it does the following:

3. Bind the name `num` to *N* in the originally active namespace.

Slicing a sequence

seq[start : end : step]

Some examples:

```
>>> values = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> values[3:]
[3, 4, 5, 6, 7, 8, 9]
>>> values[3:5]
[3, 4]
>>> values[-1]
9
>>> values[::2]
[0, 2, 4, 6, 8]
```

List comprehensions

A **list comprehension** is a short way to describe a list of values. It is syntactic sugar for an imperative loop that builds up a list one element at a time. It is analogous to functional-programming constructs such as `map` and `filter`.

Good programming practice: use a list comprehension when it can replace a single, non-nested loop or a call to `map` or `filter`. You can also use a list comprehension to replace nested loops; but you should do so only if the list comprehension is clearer than the loops.

List comprehensions can replace loops / `map`s.

Code to read all the lines from a data file, removing the newline character at the end of each line.

```
lines = open('data.txt').readlines()

# loop
data = []
for line in lines:
  data.append(line[:-1])

# map
data = list(map(lambda line: line[:-1], lines))

# list comprehension (preferred)
data = [line[:-1] for line in lines]
```

List comprehensions can replace loops / `filter`s.

Code to filter all the positive values from a list of values.

```
# loop
positiveValues = []
for value in values:
  if value > 0:
    positiveValues.append(value)

# filter
data = list(filter(lambda value: value > 0, values))

# list comprehension (preferred)
data = [value for value in values if value > 0]
```