Python Built-in Data Structures, Functions, and Files

Part 6

Functions

Part 2

Currying: Partial Argument Application

- *Currying* is computer science jargon (named after the mathematician Haskell Curry) that means deriving new functions from existing ones by *partial argument application*.
- For example, suppose we had a trivial function that adds two numbers together:
  - ```
    def add_numbers(x, y):
        return x + y
    ```
- Using this function, we could derive a new function of one variable, `add_five`, that adds `5` to its argument:
  - `add_five = lambda y: add_numbers(5, y)`
- The second argument to `add_numbers` is said to be curried.
- There’s nothing very fancy here, as all we’ve really done is define a new function that calls an existing function.
    ```

- The built-in `functools` module can simplify this process using the `partial` function:

```
In [142]: from functools import partial

          def add_numbers(x, y):
              return x + y

          add_five = partial(add_numbers, 5)
          add_five(10)

Out[142]: 15
```

Generators

- Having a consistent way to iterate over sequences, like objects in a list or lines in a file, is an important Python feature.
- This is accomplished by means of the *iterator protocol*, a generic way to make objects iterable.
- For example, iterating over a dict yields the dict keys:

```
In [143]: some_dict = {'a': 1, 'b': 2, 'c': 3}
          for key in some_dict:
              print(key)

a
b
c
```

- When you write `for key in some_dict`, the Python interpreter first attempts to create an iterator out of `some_dict`:

```
In [144]: some_dict
Out[144]: {'a': 1, 'b': 2, 'c': 3}

In [145]: dict_iterator = iter(some_dict)
          dict_iterator
Out[145]: <dict_keyiterator at 0x7f5b1d3d6638>
```

- An iterator is any object that will yield objects to the Python interpreter when used in a context like a for loop.
- Most methods expecting a list or list-like object will also accept any iterable object.
- This includes built-in methods such as `min`, `max`, and `sum`, and type constructors like `list` and `tuple`:

```
In [146]: list(dict_iterator)

Out[146]: ['a', 'b', 'c']
```

- A generator is a concise way to construct a new iterable object.
- Whereas normal functions execute and return a single result at a time, generators return a sequence of multiple results lazily, pausing after each one until the next one is requested.
- To create a generator, use the `yield` keyword instead of `return` in a function:

```
In [147]: def squares(n=10):
              print('Generating squares from 1 to {0}'.format(n ** 2))
              for i in range(1, n + 1):
                  yield i ** 2
```

- When you actually call the generator, no code is immediately executed:

```
In [148]: gen = squares()
          gen
Out[148]: <generator object squares at 0x7f5b1d4cdde0>
```

- It is not until you request elements from the generator that it begins executing its code:

```
In [149]: for x in gen:
              print(x, end=' ')

Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

Generator expresssions

- Another even more concise way to make a generator is by using a generator expression.
- This is a generator analogue to list, dict, and set comprehensions; to create one, enclose what would otherwise be a list comprehension within parentheses instead of brackets:

```
In [150]: gen = (x ** 2 for x in range(100))
          gen
Out[150]: <generator object <genexpr> at 0x7f5b1d4cdd68>
```

- Generator expressions can be used instead of list comprehensions as function arguments in many cases:

```
In [151]: sum(x ** 2 for x in range(100))
Out[151]: 328350

In [152]: dict((i, i **2) for i in range(5))
Out[152]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

itertools module

- The standard library `itertools` module has a collection of generators for many common data algorithms.
- For example, `groupby` takes any sequence and a function, grouping consecutive elements in the sequence by return value of the function.
- Here's an example:

```
In [153]: import itertools
          first_letter = lambda x: x[0]
          names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']
          for letter, names in itertools.groupby(names, first_letter):
              print(letter, list(names)) # names is a generator

A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

Function	Description
`combinations(iterable, k)`	Generates a sequence of all possible `k`-tuples of elements in the iterable, ignoring order and without replacement (see also the companion function `combinations_with_replacement`)

In [5]:
```python
import itertools

t = ['A', 'B', 'C', 'D']
for c in itertools.combinations(t, 2):
    print(c)
```

```
('A', 'B')
('A', 'C')
('A', 'D')
('B', 'C')
('B', 'D')
('C', 'D')
```

In [6]:
```python
t = ['A', 'B', 'C', 'D']
for c in itertools.combinations_with_replacement(t, 2):
    print(c)
```

```
('A', 'A')
('A', 'B')
('A', 'C')
('A', 'D')
('B', 'B')
('B', 'C')
('B', 'D')
('C', 'C')
('C', 'D')
('D', 'D')
```

`permutations(iterable, k)`	Generates a sequence of all possible `k`-tuples of elements in the iterable, respecting order

```
In [7]: t = ['A', 'B', 'C', 'D']
        for c in itertools.permutations(t, 2):
            print(c)

('A', 'B')
('A', 'C')
('A', 'D')
('B', 'A')
('B', 'C')
('B', 'D')
('C', 'A')
('C', 'B')
('C', 'D')
('D', 'A')
('D', 'B')
('D', 'C')
```

`groupby(iterable[, keyfunc])` Generates `(key, sub-iterator)` for each unique key

- Generally, the iterable needs to already be sorted on the same key function.

```
In [9]: first_letter = lambda x: x[0]
        names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']
        names.sort(key=first_letter)
        for letter, names in itertools.groupby(names, first_letter):
            print(letter, list(names))

A ['Alan', 'Adam', 'Albert']
S ['Steven']
W ['Wes', 'Will']
```

`product(*iterables, repeat=1)`	Generates the Cartesian product of the input iterables as tuples, similar to a nested `for` loop

```
In [10]: t = ['A', 'B', 'C', 'D']
         for c in itertools.product(t, repeat=2):
             print(c)

('A', 'A')
('A', 'B')
('A', 'C')
('A', 'D')
('B', 'A')
('B', 'B')
('B', 'C')
('B', 'D')
('C', 'A')
('C', 'B')
('C', 'C')
('C', 'D')
('D', 'A')
('D', 'B')
('D', 'C')
('D', 'D')
```

In [11]:
```python
t1 = ['A', 'B', 'C', 'D']
t2 = [1, 2, 3]
for c in itertools.product(t1, t2):
    print(c)
```

```
('A', 1)
('A', 2)
('A', 3)
('B', 1)
('B', 2)
('B', 3)
('C', 1)
('C', 2)
('C', 3)
('D', 1)
('D', 2)
('D', 3)
```

Errors and Exception Handling

- Handling Python errors or *exceptions* gracefully is an important part of building robust programs.
- In data analysis applications, many functions only work on certain kinds of input.
- As an example, Python’s `float` function is capable of casting a string to a floating-point number, but fails with `ValueError` on improper inputs:

```
In [154]: float('1.2345')

Out[154]: 1.2345

In [155]: float('something')

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-155-2649e4ade0e6> in <module>
----> 1 float('something')

ValueError: could not convert string to float: 'something'
```

- Suppose we wanted a version of `float` that fails gracefully, returning the input argument.
- We can do this by writing a function that encloses the call to `float` in a `try/except` block:

```
In [156]: def attempt_float(x):
              try:
                  return float(x)
              except:
                  return x
```

- The code in the `except` part of the block will only be executed if `float(x)` raises an exception:

```
In [157]: attempt_float('1.2345')
Out[157]: 1.2345

In [158]: attempt_float('something')
Out[158]: 'something'
```

- You might notice that `float` can raise exceptions other than `ValueError`:

```
In [159]: float((1, 2))

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-159-82f777b0e564> in <module>
----> 1 float((1, 2))

TypeError: float() argument must be a string or a number, not 'tuple'
```

- You might want to only suppress `ValueError`, since a `TypeError` (the input was not a string or numeric value) might indicate a legitimate bug in your program.
- To do that, write the exception type after `except`:

```
In [160]: def attempt_float(x):
              try:
                  return float(x)
              except ValueError:
                  return x
```

- We have then:

```
In [161]: attempt_float((1, 2))

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-161-8b0026e9e6b7> in <module>
----> 1 attempt_float((1, 2))

<ipython-input-160-6209ddecd2b5> in attempt_float(x)
      1 def attempt_float(x):
      2     try:
----> 3         return float(x)
      4     except ValueError:
      5         return x

TypeError: float() argument must be a string or a number, not 'tuple'
```

- You can catch multiple exception types by writing a tuple of exception types instead (the parentheses are required):

```
In [162]: def attempt_float(x):
              try:
                  return float(x)
              except (TypeError, ValueError):
                  return x

In [163]: attempt_float((1, 2))

Out[163]: (1, 2)
```

- In some cases, you may not want to suppress an exception, but you want some code to be executed regardless of whether the code in the `try` block succeeds or not.
- To do this, use `finally`:
  - ```
    f = open(path, 'w')
    try:
        write_to_file(f)
    finally:
        f.close()
    ```
- Here, the file handle `f` will *always* get closed.
    ```

- Similarly, you can have code that executes only if the `try:` block succeeds using `else:`
  - ```
    f = open(path, 'w')
    try:
        write_to_file(f)
    except:
        print('Failed')
    else:
        print('Succeeded')
    finally:
        f.close()
    ```
    ```

Exceptions in IPython

- If an exception is raised while you are `%run`-ing a script or executing any statement, IPython will by default print a full call stack trace (traceback) with a few lines of context around the position at each point in the stack:

```
In [164]: %run examples/ipython_bug.py

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
~/pydata-book-2nd-edition/examples/ipython_bug.py in <module>
     13     throws_an_exception()
     14
---> 15 calling_things()

~/pydata-book-2nd-edition/examples/ipython_bug.py in calling_things()
     11 def calling_things():
     12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()

~/pydata-book-2nd-edition/examples/ipython_bug.py in throws_an_exception()
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():

AssertionError:
```

- Having additional context by itself is a big advantage over the standard Python interpreter (which does not provide any additional context).
- You can control the amount of context shown using the `%xmode` magic command, from `Plain` (same as the standard Python interpreter) to `Verbose` (which inlines function argument values and more).