List Comprehensions, Generator Expressions & Functions

Michael Foord

https://agileabstractions.com

- Python Trainer
- Core Python Developer
- Author of IronPython in Action
- Creator of unittest.mock
- Agile, DevSecOps and design thinking
- OOP Theory
- Twitter: @voidspace

From the Agile Manifesto: “*Working software is the primary measure of progress.*”

for and tuples

- You can have multiple iteration variables

```
points = [
    (1, 4),(10, 40),(23, 14),(5, 6),(7, 8)
]

for x, y in points:
    # Loops with x = 1, y = 4
    #            x = 10, y = 40
    #            x = 23, y = 14
    #            ...
```

tuples are exanded

- Here each tuple is <u>unpacked</u> into a set of iteration variables

enumerate() function

- enumerate(*sequence* [, *start=0*])
- Provides a loop counter

```
names = ['Elwood', 'Jake', 'Curtis']
for i, name in enumerate(names):
    # Loops with i = 0, name = 'Elwood'
    #            i = 1, name = 'Jake'
    #            i = 2, name = 'Curtis'
    ...
```

- Example: keeping track of line number

```
with open(filename) as f:
    for lineno, line in enumerate(f, start=1):
        ...
```

enumerate() function

- enumerate() is a nice shortcut

```
for i, x in enumerate(s):
    statements
```

- Compare to:

```
i = 0
for x in s:
    statements
    i += 1
```

- Less typing and enumerate() runs slightly faster

zip() function

- Makes an iterator that combines sequences

```
columns = ['name', 'amount', 'price']
values = ['chair', 100, 490.1 ]

pairs = zip(a, b, strict=True)
# ('name','chair'), ('amount',100), ('price',490.1)
```

- To get the result, you must iterate

```
for name, value in pairs:
    ...
```

- Common use: making dictionaries

```
d = dict(zip(columns, values, strict=True))
```

The strict keyword requires all the input iterables to be the same length, or zip will raise an exception to warn you. Always use `strict=True`.

List Comprehensions

- Creates a new list by applying an operation to each element in a sequence

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [2*x for x in a]
>>> b
[2, 4, 6, 8, 10]
```

- Another example:

```
>>> names = ['Elwood', 'Jake']
>>> a = [name.lower() for name in names]
>>> a
['elwood', 'jake']
```

List Comprehensions

- A list comprehension can also filter

```
>>> a = [1, -5, 4, 2, -2, 10]
>>> b = [2*x for x in a if x > 0]
>>> b
[2, 8, 4, 20]
```

- Another example

```
>>> f = open('inventory.csv', 'r')
>>> chairs = [line for line in f if 'chair' in line]
```

List Comprehensions

- General syntax

```
[expression for names in sequence if condition]
```

- List comprehensions come from maths

```
a = { x 2 | x ∈ s, x > 0 }      # Math
```

- What it means

```
result = []
for names in sequence:
    if condition:
        result.append(expression)
```

- Can be used anywhere a sequence is expected

```
>>> a = [1, 2, 3, 4]
>>> sum([x*x for x in a])
30
```

List Comp: Examples

- List comprehensions are hugely useful
- Collecting the values of a specific field

```
item_names = [i['name'] for i in items]
```

- Performing database-like queries

```
a = [i for i in items if i['price'] > 100
                        and i['amount'] > 50 ]
```

- Data reductions over sequences

```
cost = sum([i['amount']*i['price'] for i in items])
```

Dictionary & Set Comprehensions

- Similar syntax can create sets

```
>>> names = { i['name'] for i in items }
```

- And dictionaries

```
>>> row = { key: value for key, value in zip(headers,
values, strict=True)}
```

- And nested syntax for loops within loops

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
odd_numbers = [
    element for row in matrix for element in row if element % 2
]
print(odd_numbers)
```

Generator Expressions

- List comprehensions are "eager", they consume their input and produce a list
- Many of the builtin functions in Python are "lazy", they produce iterable objects instead of executing immediately

```
>>> range(100)
range(0, 100)
>>> zip(['name', 'shares', 'prices'], ['GOOG', 100, 490.10])
<zip object at 0x80c0>
>>> enumerate(nums)
<enumerate object at 0x8840>
```

- Generator expressions are a lazy version of list comprehensions

Generator Expressions

- Generator expressions produce "one shot" generators
- The syntax is very similar to list comprehensions

```
>>> a = [2, 4, 6, 8, 10]
>>> b = (x**2 for x in a)
>>> b
<generator object <genexpr> at 0xf760>
>>> for result in b:
...     print(result)
...
4
16
36
64
100
```

- They don't produce a list, so the whole result set doesn't need to be in memory
- They can't be reused

Generator Expressions

- General syntax (very similar to list comprehensions)

```
(expression for names in iterable if conditional)
```

- They look better than list comprehensions in function calls

```
sum(x*x for x in a)
```

- Can be applied to any iterable and even chained together

```
>>> a = [1,2,3,4]
>>> b = (x*x for x in a)
>>> c = (-x for x in b)
>>> for i in c:
...     print(i, end=' ')
...
-1 -4 -9 -16
```

Functions

Default Arguments

- Sometimes you want an optional argument

```
def read_prices(filename, debug=False):
    ...
```

- If a default value is assigned, the argument is optional in function calls

```
d = read_prices('prices.csv')
e = read_prices('prices.dat', True)
```

- Note: arguments with defaults must appear at the end of the argument list (all required arguments go first)

Calling a Function

- Consider a simple function

```
def read_prices(filename, debug):
    ...
```

- Calling with "positional" args

```
prices = read_prices('prices.csv', True)
```

- Calling with "keyword" arguments

```
prices = read_prices(filename='prices.csv',
                     debug=True)
```

- Calling with mixed arguments

```
prices = read_prices('prices.csv', debug=True)
```

Optional/Keyword Arguments

- Arguments with default values are useful for functions that have optional features/flags

```
def parse_data(data, debug=False, ignore_errors=False):
    ...
```

- Compare and contrast calling styles:

```
parse_data(data, False, True)          # ?????

parse_data(data, ignore_errors=True)
parse_data(data, debug=True)
parse_data(data, debug=True, ignore_errors=True)
```

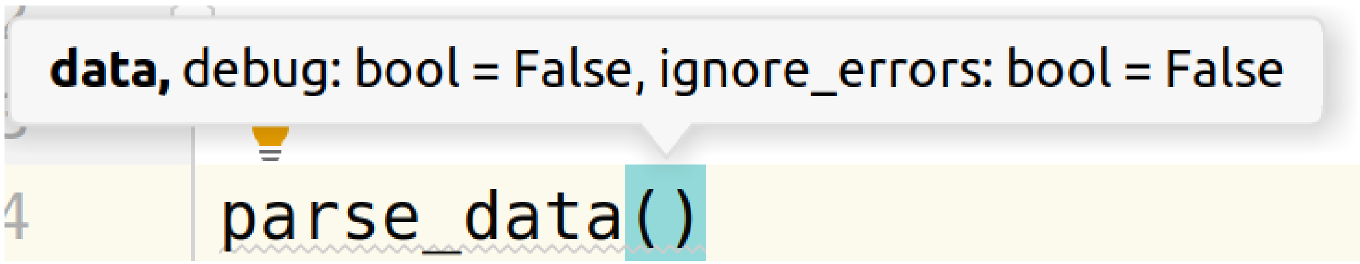
- Keyword arguments improve code clarity
- Optional arguments can be added to functions without breaking existing uses (backwards compatibility)

Design Tip

- Always give short meaningful names to function arguments
- The argument names are part of the API of the function, a design consideration
- Someone using a function may want to use the keyword calling style

```
d = read_prices('prices.csv', debug=True)
```

- Python development tools will show the names in help features and documentation



Return Values

- return returns a value

```
def square(x):
    return x*x
```

- return without a value returns None

```
def bar(x):
    statements
    return

a = bar(4)      # a = None
```

- A function without an explicit return, returns None

```
def foo(x):
    statements
    statements

a = foo(9)      # a = None
```

Multiple Return Values

- A function may return multiple values by returning a tuple

```
def divide(a,b):
    q = a // b      # Quotient
    r = a % b       # Remainder
    return q, r     # Return a tuple
```

- Usage examples:

```
x, y = divide(37, 5)        # x = 7, y = 2

x = divide(37, 5)           # x = (7, 2)
```

- Unpacking the returned tuple in the call looks like multiple return values

Positional and Keyword Only Arguments

- Python function signatures are now very rich
- We can now express positional and keyword only arguments
- Positional only arguments (mostly for compatibility with C functions) added in Python 3.8
- Keyword only arguments were new in Python 3.0

```
>>> def foo(data, /, *, debug=False):
...     pass
...
>>> foo(1, debug=True)
>>> foo(data=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got some positional-only arguments passed
as keyword arguments: 'data'
>>> foo(3, False)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
   TypeError: foo() takes 1 positional argument but 2 were
                                    given
```

Understanding Variables

- Programs assign values to variables

```
x = value        # Global variable

def foo():
    y = value    # Local variable
```

- Variable assignments occur outside and inside function definitions
- Variables defined outside a function are "global"
- Variables defined inside a function are "local"

Local Variables

- Variables inside functions are private

```
def read_portfolio(filename):
    portfolio = []
    with open(filename) as f:
        for line in f:
            fields = line.split()
            s = (fields[0], int(fields[1]), 
float(fields[2]))
            portfolio.append(s)
    return portfolio
```

- The names are not available after the function call

```
>>> stocks = read_portfolio('stocks.dat')
>>> fields
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fields' is not defined
```

- Local variables don't conflict with variables elsewhere

Global Variables

- Functions can access the values of globals

```
name = 'Dave'

def greeting():
    print('Hello', name)
```

- A quirk: functions can't modify globals

```
def spam():
    name = 'Guido'

spam()
print(name)      # prints 'Dave'
```

- All assignments inside a function create local variables

Modifying Globals

- If you must modify a global variable you declare it in the function

```
switch = False

def toggle():
    global switch
    switch = not switch   # Changes the global variable
```

- global declaration must occur before use
- Global variables are considered "bad practise" (but common in scripts)
- Avoid globals if you can (use a class instead)

Argument Passing

- When you call a function, the argument variables are names for passed values
- If mutable data types are passed (e.g. lists, dicts), they can be modified "in-place"

```python
def foo(items):
    items.append(42)

a = [1, 2, 3]
foo(a)
print(a)        # [1, 2, 3, 42]
```

- Key point: the function doesn't receive a copy (it gets a new reference to the object)

Mutable Defaults

- Don’t use mutable objects as default arguments
- The default is bound at function definition
- So that one object is shared by all function calls

```
def function(arg=[]):
    arg.append(1)
```

- Use a sentinel, or None, and create the mutable object inside the function.

```
def function(arg=None):
    if arg is None:
        arg = []
    arg.append(1)
```