Be Pythonic

Shalabh Chaturvedi, 12:30AM on 20 Nov, 2009

This article is intended for new users of Python.

When going from one language to another, some things have to be *unlearned* (see Transfer of Learning). What you know from other languages may not be always useful in Python. This page contains some idioms used in Python that I particularly like, and I hope others find useful in their quest for Pythonicity.

You need counters rarely, and iterators only occasionally

Wrong:

```
i = 0
while i<10:
     do_something(i)
     i += 1
```

Pythonic:

```
for i in xrange(10):
    do_something(i)
```

The following example indexes a list.

Wrong:

```
i = 0
while i<len(L):
    do_something(L[i])
    i += 1
```

Pythonic:

```
for item in L:
    do_something(item)
```

An iterator variable is useful when you want to maintain looping state between two 'runs':

```
itrL = iter(L)

for item in itrL:
    do_something(item)
    if is_some_condition(item):
        break

for item in itrL:    # continues where previous loop left off
    do_something_else(item)
```

You may not need that `for` loop

Python provides many higher level facilities to operate on sequences, such as `zip()`, `max()`, `min()`, list comprehensions, generator expressions and so on. See Built-in Functions for these functions and more.

Keep data in tuples, lists and dictionaries, and operate on entire collections for that fuzzy Pythonic feeling. For example, here is some code that reads a CSV file (with first row being the field names), converts each line into a dictionary record, and calculates the sum on the 'quantity' column:

```
f = open('filename.csv')                    # f is an iterator
field_names = f.next().split(',')         # get the first item from the iterator using next()
records = [dict(zip(field_names, line.split(',')))  for line in f]  # this will pull remaining
print sum(int(record['quantity'])  for record in records)
```

Though a little naive (you should be using the `csv` module anyway, which is part of the Python Standard Library), this example demonstrates some useful features. Using `zip()` with `dict()` you can combine a tuple of field names with a tuple of values and make a dictionary - combine with list comprehensions you can do this to an entire list in one go.

Tuples are not read-only lists

Tuples usually indicate a *heterogenous* collection, for example `(first_name, last_name)` or `(ip_address, port)`. Note that the *type* may be same (as in `first_name` and `last_name` may both be strings), but the real world meaning is usually different. You can think of a tuple as a row in a relational database - in fact the

database row is even called a tuple in formal descriptions of the relational model. By contrast, a list of names is always a list, even though a particular function may not change it, that does not make it a tuple.

Tuple unpacking is a useful technique to extract values from a tuple. For example:

```
for (ip_address, port) in all_connections:
    if port<2000:
        print 'Connected to %s on %s' % (ip_address, port)
```

Reading this code tells you that `all_connections` is a list (or iterable) contaning tuples of the form `(ip_address, port)`. This is much clearer than using `for item in all_connections` and then poking inside `item` using `item[0]` or similar techniques.

Unpacking is also useful while returning multiple values from a function:

```
name, ext = os.path.splitext(filename)    # split a file name into first part and extension
```

Classes are not for grouping utility functions

C# and Java can have code only within classes, and end up with many *utility* classes containing only static methods. A common example is a math functions such as `sin()`. In Python you just use a module with the top level functions.

Say no to getter and setter methods

Yes, *encapsulation* is important. No, getters and setters are not the only way to implement encapsulation. In Python, you can use a `property` to replace a member variable and completely change the implementation mechanism, with *no change* to any calling code.

Functions are objects

A function is a object that happens to be callable. This example sorts a list of dictionaries based on the value of 'price' key in the dictionaries:

```
#  define a function that returns useful data from within an object
def get_price(ob):
    return ob['price']

L.sort(key=get_price)   # sort a list using the ['price'] value of objects in the list
```

You can also use `sorted(L, key=get_price)` to return a new list instead of modifying the list in-place.

Related Links

- Python is not Java
- What is Pythonic

This article originally appeared at http://shalabh.infogami.com/Be_Pythonic2

Comments

No Comments.

Post Comment

Sign In or provide:

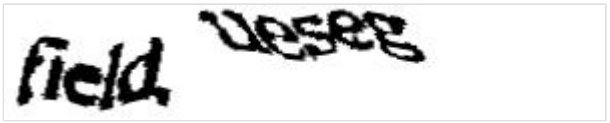
Name*

Anonymous

Email*

Not disclosed

Human Test*



Comment*

Markdown formatting

Preview Post