SIMPLE IS BETTER THAN COMPLEX

Pyconng 2018

Presented by Kelvin Adigwu @ pyconng 2018

@kelvin_adigwu

Kelvin Adigwu

Speaker Profile

>>> PYTHON/DJANGO DEVELOPER,

>>> NYSC INTERN AT LS SCIENTIFIC LIMITED, LAGOS NIGERIA.

>>> REMOTE SOFTWARE ENGINEER AT AYULLA LIMITED

>>> GRADUATE OF PHYSICS AT DELTA STATE UNIVERSITY, ABRAKA

Aim Of This Talk

This talk aims at bringing us back to python's original philosophy of being easy to read. It's title, "Simple is better than complex" is drawn from line 3 of 'The Zen of Python, by Tim Peters'

(type 'import this' on IDLE)

Target Audience

- Beginners
- Intermediate
- Non core python developers

Importance of code simplicity

- Programs must be written for people to read, and only incidentally for machines to execute.“

- Abelson & Sussman, SICP

- Makes team work easier
- Makes debugging easier
- Some idioms can be faster or use less memory than their “non-idiomatic” counterparts.

Don't reinvent the wheel!

Before writing any code,

- Check Python's standard library,
- Check the Python Package Index
- Check for existing solutions by seasoned programmers

Assigning Multiple Variables

Not so good

```
a = 1
b = 2
c = 3
my_list = [1, 2, 3]
a = my_list[0]
b = my_list[1]
 c = my_list[2]
x = 'bar'
y = 'bar'
z = 'bar'
```

Pythonic

```
a, b, c = 1, 2, 3
my_list = [1, 2, 3]
a, b, c = my_list

x = y = z = 'bar'
```

Python's Tenary style Operator

```
>>> some_boolean = True
```

Normal

```
If some_boolean:
    value = 1
else:
    value = 0
print(value)
1
```

Simpler

```
value = 1 if some_boolean else 0
print(value)
1
```

Python variables are not boxes

Unlike in C, Assigning more than one variable to same object doesn't create a copy of the object, Only creates a reference.

EX:

```
>>>a = [1, 2, 3]
>>>b = a
>>>b.append(4)
>>>print(a)
 [1, 2, 3, 4]
```

Use '==' to check for equality, not 'is'

Python's 'is' keyword is not a replacement of the equality operator(==) as it is often used. It only checks if two variables points to same object.

```
>>> a = 'Hello world!'
>>> b = 'Hello world!'
>>> c = b
>>> a == b
True
>>> a is b
False
>>> c is b
True
```

Conditionals

Normal

```
if len(a_list) > 0:
    do stuff
If number > 0:
    do stuff
```

Pythonic

```
if a_list:
    do stuff
If number:
    do stuff
```

- Checking for truth in this manner clearly shows the code's intention rather than drawing attention to the outcome of the condition

The following evaluates to True:

- True
- Numbers other than zero
- Non empty string
- Non Empty Collections

```
if 1:
if 'string is not empty':
If {'key': value}:
If [item]:
```

Are valid python statements

Multi line strings

```
>>> My_string = (“This very long string”
                 “Has been broken down”
                 “Into multiple lines”)

>>> Print(my_string)
This very long string Has been broken down Into multiple lines
```

String Formatting

Not so good

```
def user_info(user):
    return 'Name: ' + user.name + ' Age: '+ user.age
```

Pythonic

```
def user_info(user):
    return 'Name: {user.name} Age: {user.age}'.format(user=user)
```

Building Strings from Substrings

```
>>> colors = [ ‘red’, ‘blue’ ‘green’ ‘black’, ‘orange’, ‘yellow’ ]
```

Wrong

```
result = ‘ ‘
for s in colors:
    result  +=  s
```

Pythonic

```
result = ''.join(colors)
```

Use return to evaluate simple expressions

Not so good

```
def add_nums(num1, num2):
    result = num1 + num2
    return result
```

Pythonic

```
def add_nums(num1, num2):
    return num1 + num2
```

Looping with indices


```
>>> colors = [ ‘red’, ‘blue’ ‘green’ ‘black’, ‘orange’, ‘yellow’ ]
```

Wrong

```
for i in range(len(colors)):
    print( i, ‘-->’ + colors[ i ] )
```

```
0 -->red
1 -->blue
2 -->green
3 -->black
4 -->orange
5 -->yellow
```

Pythonic

```
for i, color in enumerate(colors):
    print( i, ‘-->’  +colors[ i ] )
```

```
0 -->red
1 -->blue
2 -->green
3 -->black
4 -->orange
5 -->yellow
```

Looping over two collections

```
>>> colors = [ 'red', 'blue' 'green' ]
>>> names = [ 'kelvin', 'john', 'paul', 'titus' ]
```


Wrong

```
n = min( len(names), len(colors))
for i in range(n):
    print(names[ i ], '-->' + colors[ i ])
```

```
kelvin -->red
john -->blue
paul -->green
titus -->black
```

Pythonic

```
for name, color in zip(names, colors):
    print(name, '-->' +color)
```

```
kelvin -->red
john -->blue
paul -->green
titus -->black
```

List Comprehensions

Python's method of making a new list from a collection

The traditional way, with for and if statements:

```
new_list = [ ]
for item in a_list:
    if condition(item):
        new_list.append(fn(item))
```

With List comprehension

```
new_list = [ fn( item ) for item in a_list
if condition( item ) ]
```

EX:

```
>>> [n ** 2 for n in range(10) ]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Use a list comprehension when a computed list is the desired end result.
- Use a generator expression(replacing [ ] with ( )) when the computed list is just an intermediate step.

Just because you can do everything with listcomps doesn't mean you should

- The conditions becomes more complex
- Deeply nested objects are involved
- You don't actually want a list

EX: Wrong:

```
[team.set_location(HOME) for team in teams if team in home_teams]
```

Better:

```
for team in league_teams:
    if team in home_teams_today:
        team.set_location(HOME)
```

Leverage on the power of Sets

If A and b are two sets:

Difference: The set of elements in A but not in B

Intersection: The set of elements in both A and B

Difference: The set of elements in A but not in B(order matters)

Symmetric Difference: The set of elements in either A or B but not both A and B

Any iterable can be converted to set. Simply pass them as arguments to set()

Example of sets in operation

```
>>> ls1 = range(5)
>>> ls2 = range(10)
```

Not so good

```
common = [ ]
for element in ls1:
    if element in ls2:
        common.append(element)
print(common)
```

[0, 1, 2, 3, 4]

Pythonic

```
common = list( set(ls1) & set(ls2) )
print(common)
```

[0, 1, 2, 3, 4]

EAFP vs. LBYL

It's **E**asier to **A**sk **F**orgiveness than **P**ermission. Perform an action, and fall back to a default if it fails. As opposed to it's counterpart:

Look **B**efore **y**ou **L**eap

EX: If x must be a string for your code to work, it is better to call

```
str(x)
```

and catching the error if it fails, than doing

```
if isinstance(x, str):
    str(x)
```

EAFP vs. LBYL Example: counting with dictionaries

```
colors = [ 'red', 'blue' 'green' 'black', 'orange', 'yellow' ]
```


LBYL

```
d = { }
for color in colors:
    if color not in d:
        d[color] = 0
    d[color] += 1
```

EAFP

```
d = { }
for color in colors:
    d[color] = d.get(color, 0) + 1
```

NB: an even better approach is using default dict

Use decorators to factor out business logic from administrative logic

Combined function

```
def web_lookup(url, saved={ }):
    if url in saved:
        return saved[url]
    page = requests.get(url)
    saved[url] = page
    return page
```

With decorators

```
@cache
def web_lookup(url):
    return requests.get(url)
```

General Notes

- Clarify function calls with keyword arguments
- Return named tuples for multiple outputs
- Good naming is essential
- Use Backslash to Continue Statements
- Catch specific errors in a try- except block
- Never bar an exception, it makes debugging harder for someone else
- Avoid wildcard imports
- Avoid the use of one time variables
- Use _ for necessary throw away variables

print('Thank You!)

Pyconng 2018