Way

Idiomatic Python. Coding the smart way.

Seremba John Paul Follow
Sep 5, 2017 · 5 min read

Let your code be smarter than your desk.

The key characteristic of python is readability. It is of great importance that we leverage on readability, since code is read much more often than it is written.

Jeff Knupp in his book “Writing Idiomatic Python”, states that;

"We may docume

old code reviews three times a day, changes, the code is king."

Python has code style guidelines and idioms. Idioms in a programming language let future readers know exactly what we're trying to do.

I have listed 15 tips that will help you become a smart python programmer.

1. Chained comparison operators

Comparisons can be chained arbitrarily. It makes the statements more concise and also has a positive effect on performance.

Bad

```
if x <= y and y <= z:
  # do something
```

Good

```
if x <= y <= z:
  # do something
```

2. Indentation

Avoid placing conditional branch code on the same line as the colon. Python uses indentation to indicate scope, and it makes it easy to determine what will be executed as part of a conditional statement.

Bad

```
name = 'John'; address = 'Kampala'

if name: print(name)

print(address)
```

Good

```
name = ‘John’
address = ‘Kampala’

if name:
  print(name)

print(address)
```

3. Use the Falsy & Truthy Concepts

One thing to avoid is comparing directly to `True`, `False` or `None`. The Term “*truthy*” refers to values that shall always be considered true, and “*falsy*” refers to values that shall always be considered false.

For example an empty list/sequences `[]`, empty dictionaries `{}` `None`, `False`, `Zero` for numeric types, are considered “*falsy*”. On the other hand, almost everything else is considered “*truthy*”.

Bad

```
x = True
y = 0

if x == True:
  # do something
elif x == False:
  # do something else

if y == 0:
  # do something

ls = [2, 5]

if len(ls) > 0:
  # do something
```

Good

```
(x, y) = (True, 0)

# x is truthy
if x:
  # do something
else:
  # do something else

# y is falsy
if not y:
  # do something

ls = [2, 5]

if ls:
  # do something
```

4. Ternary Operator replacement

Python does not have the ternary operator (e.g: x ? True : False) that many languages have. However an alternative form may be used:

Bad

```
a = True
value = 0

if a:
  value = 1

print(value)
```

Good

```
a = True
value = 1 if a else 0

print(value)
```

5. Use the 'in' k

Checking a variable against a number of values repeatedly is unnecessarily verbose. Use a check for existence instead.

Bad

```
city = 'Nairobi'
found = False

if city == 'Nairobi' or city == 'Kampala' or city == 'Lagos':
  found = True
```

Good

```
city = 'Nairobi'
found = city in {'Nairobi', 'Kampala', 'Lagos'}
```

The `in` keyword is also useful when iterating over an iterable.

Bad

```
cities = ['Nairobi', ‘Kampala’, ‘Lagos’]
index = 0

while index < len(cities):
  print(cities[index])
  index += 1
```

Good

```
cities = [‘Nairobi’, ‘Kampala’, ‘Lagos’]

for city in cities:
  print(city)
```

6. Use 'return' values

Bad

```
def check_equal(x, y):
  result = False

  if x == y:
    result = True

  return result
```

Good

```
def check_equal(x, y):
  return x == y
```

7. Multiple assignment

Always use multiple assignment to condense variables all set to the same value. This greatly improves the readability of your code.

Bad

```
x = 'foo'
y = 'foo'
z = 'foo'
```

Good

```
x = y = z = 'foo'
```

8. Formatting Strings

The worst approach to formatting strings is to use the `+` operator to concatenate a mix of static strings and variables. However, the clearest and most idiomatic way to format strings is to use t

s placeholders with values.

Bad

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

Good

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

9. List comprehension

Use list comprehensions to create lists or a transformed version of a list. When list comprehensions are used well, they increase code clarity. There are also performance benefits that arise from list comprehensions.

Bad

```
ls = list()

for element in range(10):
  if not(element % 2):
    ls.append(element)

# We may also employ a lambda function

ls = list(filter(lambda element: not(element % 2), range(10)))
```

Good

```
ls = [element for element in range(10) if not(element % 2)]
```

10. enumerate

In scenarios where you may want to access the index of each element in a list, its better you employ the `enumerate` function.

Bad

```
ls = list(range(10))
index = 0

while index < len(ls):
  print(ls[index], index)
  index += 1
```

Good

```
ls = list(range(10))

for index, value in enumerate(ls):
  print(value, index)
```

11. Dictionary Comprehension

The list comprehension is a well known python construct. However, less is known about the `dict comprehension`. Its purpose is to construct a dictionary using a well understood comprehension syntax.

Bad

```
emails = {}

for user in users:
  if user.email:
    emails[user.name] = user.email
```

Good

```
emails = {us
```

```
.email}
```

12. Sets

Leverage on the powerful set operations. Understanding the basic mathematical set operations is the key to harnessing their power.

`Union` : The set of elements in `A` , `B` or both (written as `A | B`)

`Intersection` : The set of elements in both `A` and `B` (written as `A & B`)

`Difference` : The set of elements in `A` but not in `B` (written as `A – B`)

NB The order matters for `Difference` . `A – B` is NOT the same as `B – A` .

`Symmetric Difference` : The set of elements in either `A or B` but not both `A and B` (written as `A ^ B`)

Bad

```
ls1 = [1, 2, 3, 4, 5]
ls2 = [4, 5, 6, 7, 8]

elements_in_both = []

for element in ls1:
  if element in ls2:
    elements_in_both.append(element)

print(elements_in_both)
```

Good

```
ls1 = [1, 2, 3, 4, 5]
ls2 = [4, 5, 6, 7, 8]

elements_in_both = list( set(ls1) & set(ls2) )

print(elements_in_both)
```

13. Set Comprehension

The `set comprehension` syntax is relatively new and often overlooked. Just as lists, sets can also be generated using a comprehension syntax.

Bad

```
elements = [1, 3, 5, 2, 3, 7, 9, 2, 7]
unique_elements = set()

for element in elements:
  unique_elements.add(element)

print(unique_elements)
```

Good

```
elements = [1, 3, 5, 2, 3, 7, 9, 2, 7]
unique_elements = {element for element in elements}

print(unique_elements)
```

14. Use the default parameter of 'dict.get' to provide default values

Many times we overlook the definition of the default parameter in the usage of `dict.get()`. However in order to avoid the nasty `KeyError` exception, its good practice we provide a default value.

Bad

```
auth = None

if 'auth_token' in payload:
  auth = payload['auth_token']
else:
  auth = 'Unauthorized'
```

Good

```
auth = payload.get('auth_token', 'Unauthorized')
```

15. Don’t Repeat Yourself (DRY)

Always try not repeat code blocks as you write your code. Whenever you find yourself repeating something, think about creating helper methods/functions or even variables. DRY is a complex programming model; here is a very nice article to get you started.

Bad

```
if user:
  print('------------------------------')
  print(user)
  print('------------------------------')
```

In the example above, we have repeated `-` over 30 times which is really not good. We can improve it thus:

Good

```
if user:
  print('{0}\n{1}\n{0}'.format('-'*30, user))
```

There is still a lot to cover about idiomatic python. The knowledge I have shared is just a tip of the iceberg.

Detailed examples and explanations can be found in Jeff Knupp’s book **Writing Idiomatic Python**. You can grab yourself a copy from here.

You can follow me on Twitter.