**Ideal Code Standards:**

**1.Use dictionary instead of if..elif and handle KeyError Exception in dictionary.**

**For Example**:

if name == "John":

print "This is John, he is an artist"

elif name == "Ted":

print "This is Ted, he is an engineer"

elif name == "Kennedy":

print "This is Kennedy, he is a teacher"'

Use Dictionary

name\_job\_dict = {

"Josh": "This is John, he is an artist",

"Ted": "This is Ted, he is an engineer",

"Kenedy": "This is Kennedy, he is a teacher",

}

print name\_job\_dict[name]

**2. Use built-in functions wherever possible**

--> Built in functions like map(),min() etc are implemented in C code. So the interpreter doesn’t have to execute the loop, this gives a considerable speedup.

-->Use Built-In Operators:Python is an interpreted language and based on high-level abstractions. So you should use the built-ins wherever possible. It’ll make your code more efficient because the built-ins are pee-compiled and fast. Whereas the lengthy iterations which include interpreted steps get very slow.

**3.Try multiple coding approaches**

--> Using precisely the same coding approach every time we create an application will almost certainly result in some situations where the application runs slower than it might.

**For example** (Initializing Dictionary Elements):

**# slower**

mydict = {'g':1,'e':1,'e':1,'k':1}

word = 'geeksforgeeks'

for w in word:

if w not in mydict:

mydict[w] = 0

mydict[w] += 1

print (mydict)

**# faster**

mydict = {'g':1,'e':1,'e':1,'k':1}

word = 'geeksforgeeks'

for w in word:

try:

mydict[w] += 1

except KeyError:

mydict[w] = 1

print (mydic)

**4. Use xrange instead of range**

-->range() – This returns a list of numbers created using range() function.

-->xrange() – This function returns the generator object that can be used to display numbers only by looping. Only particular range is displayed on demand and hence called “lazy evaluation”.

**# slower**

x = [i for i in range(0,10,2)]

print(x)

**# faster**

x = [i for i in xrange(0,10,2)]

print(x)

**5.Use Python multiple assignment to swap variables**

This is elegant and faster in Python.

**Example:**

**# slower**

x = 2

y = 5

temp = x

x = y

y = temp

print( x,y)

**# faster**

x,y = 3,5

x, y = y, x

print( x,y)

**6.Use local variable if possible**

-->Python is faster retrieving a local variable than retrieving a global variable. That is, avoid the “global” keyword. So if you are going to access a method often (inside a loop) consider writing it to a variable.

**7.Limit Method Lookup In A Loop**

-->When working in a loop, you should cache a method call instead of calling it on the object. Otherwise, the method lookups are expensive.

**Example:**

**#slow**

>>> for it in xrange(10000):

myLib.findMe(it)

**#Fast**

>>> findMe = myLib.findMe

for it in xrange(10000):

findMe(it)

**8.Optimize Loop**

-->The Python engine spends substantial efforts in interpreting the for loop construct. Hence, it’s always preferable to replace them with built-in constructs like Maps.

--> Move calculations outside the loop

-->Declaration of variable should be outside the loop

--> Try to avoid use of nested loop

**For Example:**

**# Slow**

for x in a:

for y in b:

if x == y:

yield (x,y)

**# Fast** -- >return set(a) & set(b)

---> Let’s consider a function which updates the list of Zipcodes, strips the trailing spaces, and use a for loop.

newZipcodes = []

for zipcode in oldZipcodes:

newZipcodes.append(zipcode.strip())

**Example-1.**

Now, see how you can translate the above into a single line using the map object. It’ll also be more cost efficient now.

newZipcodes = map(str.strip, oldZipcodes)

**Example-2.**

We can even use the list comprehensions to make the syntax a bit more linear.

Zipcodes += [iter.strip() for iter in newZipcodes]

**Example-3.**

And lastly, the fastest approach would be to convert the for loop into a generator expression.

itertools.chain(Zipcodes, (iter.strip() for iter in newZipcodes))

**9.String concatenation**

**# slow**

msg = 'hello ' + my\_var + ' world'

**# faster**

msg = 'hello %s world' % my\_var

**# or better**

msg = 'hello {} world'.format(my\_var)

**10.Use keys for sorts**

In Python, we should use the key argument to the built-in sort instead, which is a faster way to sort.

# Python program to illustrate

# using keys for sorting

somelist = [1, -3, 6, 11, 5]

somelist.sort()

print (somelist)

s = 'geeks'

# use sorted() if you don't want to sort in-place:

s = sorted(s)

print (s)

# using operator

import operator

test = [(11, 52, 83), (61, 20, 40), (93, 72, 51)]

print("Before sorting:", test)

test.sort(key=operator.itemgetter(0))

print("After sorting[1]: ", test)

**11.Optimizing Using Strings**

String concatenation is slow, don’t ever do that inside a loop. Instead, use the Python’s join method. Or, use the formatting feature to form a unified string.Reg Ex operations in Python are fast as they get pushed back to C code. However, in some cases, basic string methods like <isalpha()/isdigit()/startswith()/endswith()> works better.

**# Slow**

s = **'hellogeeks'**slist = **''  
for** i **in** s:  
 slist = slist + i  
print(slist)

**# Fast**  
st = **'hellogeeks'**slist = **''**.join([i **for** i **in** s])  
print(slist)

**# Slow**

string = ""

for i in range(0, 256, 16): # 0, 16, 32, 48, 64, ...

s = ""

for character in map(chr, list[i:i+16]):

s = s + character

string = string + s

return string

**# Fast**

import string

return string.joinfields(map(chr, list), "")

or

import array

return array.array('B', list).tostring()

**12. Optimizing With If Statement.**

Like most programming languages allow lazy-if evaluation, so does the Python. It means, if there are joining ‘AND’ conditions, then not all conditions will be tested in case one of them turns false.

1. You can adjust your code to utilize this behavior of Python. For example, if you are searching for a fixed pattern in a list, then you can reduce the scope by adding the following condition.

Add an ‘AND’ condition which becomes false if the size of the target string is less than the length of the pattern.

Also, you can first test a fast condition (if any) like “string should start with an @” or “string should end with a dot.”.

2. You can test a condition <like if done is not None> which is faster than using <if done != None>.

**13.By using set default**, we get cleaner code:

**For Example:**

def count\_duplicates(numbers):

result = {}

for number in numbers:

result.setdefault(number, 0) # this is clearer

result[number] += 1

return result

14.Avoid repeation of same code again again , if it requires multiple time make sepearte method for it.

**Profile Your Code:**

**i-Use Stop-Watch Profiling With <Timeit>-->**It’s the traditional way of profiling using the Python’s <timeit> module. It records the time a segment of your code takes for execution. It measures the time elapsed in milliseconds.

**Example:**

import timeit

subStrings=['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']

def simpleString(subStrings):

finalString = ''

for part in subStrings:

finalString += part

return finalString

print('simpleString() Time : ' + str(timeit.timeit('simpleString(subStrings)', setup='from \_\_main\_\_ import simpleString, subStrings')))

**ii-Use Advanced Profiling With <CProfile>**

**Example:**

import cProfile

cProfile.run('10\*10')

run by this command -- > python -m cProfile -s cumtime test1.py

**How To Interpret CProfile Results**

1. <ncalls>: It is the number of calls made.

2. <tottime>: It is the aggregate time spent in the given function.

3. <percall>: Represents the quotient of <tottime> divided by <ncalls>.

4. <cumtime>: The cumulative time in executing functions and its subfunctions.

5. <percall>: Signifies the quotient of <cumtime> divided by primitive calls.

6. <filename\_lineno(function)>: Point of action in a program. It could be a line no. or a function at some place in a file.

---- <https://www.techbeamers.com/python-code-optimization-tips-tricks/#h5.1>

------<https://www.geeksforgeeks.org/optimization-tips-python-code/>