Program for prime number -bad code

```
i=25
for x in range(2, i//2+1):
      if i%x==0:
         print("The number {} is not prime".format(i))
         break
if x ==i//2:
   print ("{} is a prime number".format(i))
```

```
The number 25 is not prime
```

Good code using for... else

```
i=25
for x in range(2, i//2+1):
      if i%x==0:
         print("The number {} is not prime".format(i))
         break
else:
    print ("{} is a prime number".format(i))
```

```
The number 25 is not prime
```

Using Unpacking to Write Concise Code

Packing and unpacking are powerful Python features. You can use unpacking to assign values to your variables:

```
a, b = 2, 'my-string'
print(a)
print(b)
```

Bad unpacking

```
x = (1, 2, 4, 8, 16)
a = x[0]
b = x[1]
c = x[2]
d = x[3]
e = x[4]
print(a, b, c, d, e)
```

Excellent unpacking

```
a,b,c,d,e=x
print(a, b, c, d, e)
```

unpacking some elements

```
a, *y, e = x
print(a)
print(y)
print(e)
```

Using Chaining to Write Concise Code

Python allows you to chain the comparison operations. So, you don't have to use and to check if two or more comparisons are True:

```
x = 4
print(x >= 2 and x <= 8)
```

Instead, you can write this in a more compact form, like mathematicians do:

```
print(2 <= x <= 8)
print(2 <= x <= 3)
```

chained assignments

```
x = y = z = 2
```

```
x, y, z # when we use , it becomes tuple
```

Checking against None

```
# normal way
x, y = 2, None
print(x == None)
print(y == None)
print(x != None)
print(y != None)
```

In pythonic way

```
x is None

print(x is None)
print(y is None)
print(x is not None)
print(y is not None)
```

Iterating over Sequences

```
x = [1, 2, 4, 8, 16]
for i in range(len(x)):
  print(x[i])
```

But instead we can do like this in elegant way

```
for item in x:
  print(item)
```

to iterate in the reversed order

```
for i in range(len(x)-1, -1, -1):
  print(x[i])
```

But in elegant way!

```
for item in x[::-1]:
 print(item)
```

The Pythonic way is to use reversed to get an iterator that yields the items of a sequence in the reversed order:

```
for item in reversed(x):
  print(item)
```

Sometimes you need both the items from a sequence and the corresponding indices:

```
for i in range(len(x)):
  print(i, x[i])
```

It’s better to use enumerate to get another iterator that yields the tuples with the indices and items:

```
for i, item in enumerate(x):
 print(i, item)
```

what if you want to iterate over two or more sequences? Of course, you can use the range again:

```
y = 'abcde'
for i in range(len(x)):
  print(x[i], y[i])
```

In this case, Python also offers a better solution. You can apply zip and get tuples of the corresponding items:

```
for item in zip(x, y):
  print(item)
```

You can combine it with unpacking:

```
for x_item, y_item in zip(x, y):
  print(x_item, y_item)
```


Dictionary can be iterated in these two ways

```
z = {'a': 0, 'b': 1}
for k in z:
  print(k, z[k])
```

```
for k, v in z.items():
  print(k, v)
```

Comparing to Zero

When you have numeric data, and you need to check if the numbers are equal to zero, you can but don't have to use the comparison operators == and !=:

```
#To print only non zero values from tuple x
x = (1, 2, 0, 3, 0, 4)
for item in x:
    if item != 0:
      print(item)
```

The Pythonic way is to exploit the fact that zero is interpreted as False in a Boolean context, while all other numbers are considered as True:

bool(0)- False

bool(-1), bool(1), bool(20), bool(28.4) - (True, True, True, True)

```
for item in x:
  if item:
    print(item)
```

Avoiding Mutable Optional Arguments

```
def func(value, seq=[]):
  seq.append(value)
  return seq
print(func(value=2))
```

```
print(func(value=4))
```

```
print(func(value=6))
```

```
print(func(value=8))
```

keep away from that with some additional logic.

```
def func(value, seq=None):
  if seq is None:
    seq = []
    seq.append(value)
    return seq
print(func(value=6))
```

Using Context Managers to Release Resources

y_file = open('filename.csv', 'w')

do something with `my_file`

To properly manage the memory, you need to close this file after finishing the job:

my_file = open('filename.csv', 'w')

#do something with `my_file and`

my_file.close()

with open('filename.csv', 'w') as my_file:

#do something with `my_file<br>`

'"Using the with block means that the special methods .enter() and .exit() are called, even in the cases of exceptions.
These methods should take care of the resources. You can achieve especially robust constructs by combining the context managers and exception handling."'

If Statements

Avoid comparing directly to True , False , or None
All of the following are considered False :

None

False

zero for numeric types

empty sequences

empty dictionaries

Instead of using if foo == True:

Conclusions

This article gives several advises on how to write a more efficient, more readable, and more concise code. In short, it shows how to write a Pythonic code. In addition, PEP 8 provides the style guide for Python code, and PEP 20 represents the principles of Python language.