Python Coding Rules

Python basics

Kunal Khurana

2023-12-06

Table of contents

General 2
- Prefer Unpacking Over Indexing 3
- Prefer enumerate Over range 3
- Use zip to process Iterators in parallel 3
- Avoid 'else' Blocks After 'for' and 'while' Loops 5
- Prevent repetition with assignment Expressions such as 'warlus operator' . . . 7

Lists and dictionaries 8
- Know how to slice sequneces 8
- Avoid striding and slicing in a single expression 8
- Perfect Catch-'All Unpacking Over Slicing' 9
- Sort by Complex Criteria using the 'key' parameter 9
- Dictionaries : insertion ordering, dict types, dict values 11
- Prefer 'get' Over 'in' and 'KeyError' to handle missing dictionary keys 11
- Prefer 'defaultdict' Over 'Setdefault' to handle missing items 14

Functions 15
- Never Unpack more than 3 variables when fucntions return multiple vaues . . . 15
- Prefer raising exceptions to returning None 16
- Know how Closures Interact with Variable Scope 17
- Reduce Visual Noise with Variable Positional Arguments 18
- Provide Optional Behavior with Keyword Arguments 19
- Use None and Docstrings to Specify Dynamic Default Arguments 19
- Define Function Decorators with funtools.wraps 20

General

C-style formatting strings in Python (4 errors)

- reversing order gives traceback
- difficult to read the code
- using same value multiple times in tuple (repeat it in the right side)
- dictionary formats

Write helper functions instead of complex expressions

- Use if/else conditional to reduce visual noise
- Moreover, if/else expression provides a more readable alternative over the boolean or/and in expressions.

Prefer Unpacking Over Indexing

- use special syntax to unpack multiple values and keys in a single statement.

Prefer enumerate Over range

- range (built-in funciton) is useful for loops
- prefer enumerate instead of looping over a range

```
# example of enumeration with list-
flavor_list = ['vanilla', 'chocolate', 'pecan', 'strawberry']
for flavor in flavor_list:
    print(f'{flavor} is delicious')
```

```
vanilla is delicious
chocolate is delicious
pecan is delicious
strawberry is delicious
```

Use zip to process Iterators in parallel

```
names = ['Kunal', 'Xives', 'pricila']
counts = [len(n) for n in names]
print(counts)
```

```
[5, 5, 7]
```

```
# iterating over lenght of lists
longest_name = None
max_count = 0

for i in range(len(names)):
```

```
    count = counts[i]
    if count > max_count:
        longest_name = names[i]
        max_count = count

print(longest_name)
```

```
pricila
```

```
# we see that the above code is a bit noisy.
# to imporve it, we'll use the enumerate method

for i, name in enumerate(names):
    count = counts[i]
    if count > max_count:
        longest_name = name
        max_count = count
print(longest_name)
```

```
pricila
```

```
# to improve it further, we'll use the inbuilt zip function

for name, count in zip(names, counts):
    if count > max_count:
        longest_name = name
        max_count = count

print(longest_name)
```

```
pricila
```

```
# zip's behavior is different if counts are not updated

names.append('Rosy')
for name, count in zip(names, counts):
    print(name)
```

```
Kunal
Xives
pricila
```

```
# so, be careful when using iterators of different lenght.

# consider using zip_longest function from itertools instead

import itertools
for name, count in itertools.zip_longest (names, counts):
    print (f'{name}: {count}')
```

```
Kunal: 5
Xives: 5
pricila: 7
Rosy: None
```

Avoid 'else' Blocks After 'for' and 'while' Loops

```
# for loops first

for i in range(3):
    print('Loop', i)
else:
    print('Else block!')
```

```
Loop 0
Loop 1
Loop 2
Else block!
```

```
# using break in the code

for i in range(3):
    print('Loop', i)
    if i == 1:
        break

else:
```

```
    print('Else block!')
```

```
Loop 0
Loop 1
```

```
# else runs immediately if looped over an empty sequence

for x in []:
    print('Never runs')
else:
    print('For else block!')
```

```
For else block!
```

```
# else also runs when while loops are initially false
while False:
    print('Never runs')
else:
    print('While else block!')
```

```
While else block!
```

```
## finding coprimes (having common divisor i.e. 1)

a = 11
b = 9

for i in range(2, min(a, b) + 1):
    print ('Testing', i)
    if a% i == 0 and b%i == 0:
        print('Not coprime')
        break
else:
    print('coprime')
```

```
Testing 2
Testing 3
```

```
Testing 4
Testing 5
Testing 6
Testing 7
Testing 8
Testing 9
coprime
```

Prevent repetition with assignment Expressions such as 'warlus operator'

```python
# Without the walrus operator
even_numbers_without_walrus = []
count = 0
while count < 5:
    number = count * 2
    if number % 2 == 0:
        even_numbers_without_walrus.append(number)
        count += 1

print(even_numbers_without_walrus)
```

```
[0, 2, 4, 6, 8]
```

```python
# With the walrus operator
even_numbers_with_walrus = []
count = 0
while count < 5:
    if (number := count * 2) % 2 == 0:
        even_numbers_with_walrus.append(number)
        count += 1

print(even_numbers_with_walrus)
```

```
[0, 2, 4, 6, 8]
```

Lists and dictionaries

Know how to slice sequneces

```
#somelist [start:end]
a = ['a', 'b', 'c', 'd', 'e', 'f']
print ('Middle two: ', a[2:4])
```

```
Middle two:  ['c', 'd']
```

Avoid striding and slicing in a single expression

```
b = [1, 2, 3, 4, 5, 6]
odds = b[::2]
evens = b[1::2]
print(odds)
print(evens)
```

```
[1, 3, 5]
[2, 4, 6]
```

```
# stride syntax that can introduce bugs ; Avoid
c = b'rouge'
d = c[::-1]

print(d)
```

```
b'eguor'
```

```
x = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
print(x[2::2])     # ['c', 'e', 'g']
print(x[-2::-2])   # ['g', 'e', 'c', 'a']
print(x[-2:2:-2])  # ['g', 'e']  #[start: stop : step]
print(x[2:2:-2])   # []
```

```
['c', 'e', 'g']
['g', 'e', 'c', 'a']
['g', 'e']
[]
```

Perfect Catch-'All Unpacking Over Slicing'

- Unpacking - extracting individual elements from a sequence (like a list or tuple) and assigning them to variables.
- Slicing - selecting a subset of elements from a sequence.

```python
# Example sequence
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Using slicing to get a portion of the sequence
subset = numbers[2:8]

# Using unpacking to assign values to variables
first, *middle, last = subset   # *used for extended unpacking

# Print the results
print("Subset:", subset)
print("First element:", first)
print("Middle elements:", middle)
print("Last element:", last)
```

```
Subset: [3, 4, 5, 6, 7, 8]
First element: 3
Middle elements: [4, 5, 6, 7]
Last element: 8
```

Sort by Complex Criteria using the 'key' parameter

- sort method works for all built-in types (strings, floats, etc.), but it doesn't work for the classes, including a **repr** method for instance.

```python
class Tool:
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight
```

```
    def __repr__(self):
        return f'Tool({self.name}, {self.weight})'

# Example usage of the Tool class
tools = [
    Tool('level', 3.5),
    Tool('hammer', 1.25),
    Tool('screwdriver', 0.5),
    Tool('chisel', 0.25),
]

# tools.sort()   #this will give us a traceback

# Display the unsorted list of tools
print('Unsorted:')
for tool in tools:
    print(repr(tool))

# Sort the tools based on their names
tools.sort(key=lambda x: x.name)

# Display the sorted list of tools
print('\nSorted:')
for tool in tools:
    print(tool)
```

```
Unsorted:
Tool(level, 3.5)
Tool(hammer, 1.25)
Tool(screwdriver, 0.5)
Tool(chisel, 0.25)

Sorted:
Tool(chisel, 0.25)
Tool(hammer, 1.25)
Tool(level, 3.5)
Tool(screwdriver, 0.5)
```

Dictionaries : insertion ordering, dict types, dict values

```
# cutest baby animal

votes = {
    'otter': 1281,
    'polar bear': 587,
    'fox': 863,
}

# save the rank to an empty dictionary
def populate_ranks(votes, ranks):  #takes votes and ranks dictionary
    names = list(votes.keys())
    names.sort(key=votes.get, reverse=True)
    for i, name in enumerate(names, 1):
        ranks[name] = i

# function that returs the animal with hightest rank
def get_winner(ranks):
    return next(iter(ranks))

# results
ranks = {}
populate_ranks(votes, ranks)
print(ranks)
winner = get_winner(ranks)
print(winner)
```

```
{'otter': 1, 'fox': 2, 'polar bear': 3}
otter
```

Prefer 'get' Over 'in' and 'KeyError' to handle missing dictionary keys

- accessing and assigning
- for maintaining dictionaries, consider Counter class from the collections built-in module
- setdefault is another shortened method other than get method, but readability is not clear. so, avoid it

example 1

```
bread = {
    '14grain': 4,
    'multigrain' : 2
}

#1) 'in' method
key = 'wheat'

if key in bread:
    count = bread[key]
else:
    count = 0

bread[key] = count + 1  #incrementing the count by 1 for 'wheat' key
```

#2) 'KeyError' method key = 'wheat'

try: count = bread[key] except KeyError: count = 0

bread[key] = count + 1

```
#3) 'get' method - best one (shortest and clearest)

key = 'oats'

count = bread.get(key,0)
bread[key] = count + 1
```

```
bread
```

```
{'14grain': 4, 'multigrain': 2, 'wheat': 2, 'oats': 1}
```

example 2

```
# more complex dictionary, to know who voted for which type of bread

votes = {
    '14grain' : ['Bob', 'Ashley', 'Suzan', 'Susan'],
    'multigrain' : ['Dikshita', 'Kavya'],
```

```
    'wheat' : ['Bhavna', 'Shristi'],
    'oats' : ['Nikumbh']
}

key = 'kinoa'
who = 'Raph'

if key in votes:
    names = votes[key]
else:
    votes[key] = names = []

names.append(who)
print (votes)
```

```
{'14grain': ['Bob', 'Ashley', 'Suzan', 'Susan'], 'multigrain': ['Dikshita', 'Kavya'], 'wheat
```

```
# try except

try:
    names = votes[key]
except KeyError:
    votes[key] = names =[]

names.append(who)
print(votes)
```

```
{'14grain': ['Bob', 'Ashley', 'Suzan', 'Susan'], 'multigrain': ['Dikshita', 'Kavya'], 'wheat
```

```
# get method
names = votes.get(key)
if names is None:
    votes[key] = names = []

names.append(who)

print(votes)
```

```
{'14grain': ['Bob', 'Ashley', 'Suzan', 'Susan'], 'multigrain': ['Dikshita', 'Kavya'], 'wheat
```

```
# prevent repetition
if (names := votes.get(key)) is None:
    votes[key] = names = []
names.append(who)

print(votes)
```

```
{'14grain': ['Bob', 'Ashley', 'Suzan', 'Susan'], 'multigrain': ['Dikshita', 'Kavya'], 'wheat
```

Prefer 'defaultdict' Over 'Setdefault' to handle missing items

```
# list of countires and cities visited
visits = {
    'India' : {'Punjab', 'Rajastan', 'Goa', 'Himachal Pardesh', 'Haryana'},
    'UAE' : {'Dubai'},
    'Nepal' : {'Kathmandu'},
    'Canada' : {'Québec', 'Ontario'},
}


# using setdefalut method to add to the list (method 1)

visits.setdefault('France', set()).add('Remi')  #short

if (japan := visits.get('Japan')) is None:      #long
    visits['Japan'] = japan = set()
japan.add('Kyoto')

print(visits)
```

```
{'India': {'Rajastan', 'Haryana', 'Punjab', 'Himachal Pardesh', 'Goa'}, 'UAE': {'Dubai'}, 'N
```

```
# how about i create a class then add places

from collections import defaultdict

class Visits:
    def __init__(self):
       self.data = defaultdict(set)
```

```
    def add(self, country, city):
        self.data[country].add(city)

visits = Visits()
visits.add('England', 'Bath')
visits.add('England', 'London')
print(visits.data)
```

```
defaultdict(<class 'set'>, {'England': {'Bath', 'London'}})
```

Functions

Never Unpack more than 3 variables when fucntions return multiple vaues

```
# Function returning multiple values
def get_person_details():
    name = "John"
    age = 30
    city = "Montréal"
    gender = "Male"
    return name, age, city, gender

# Unpacking with three variables
name, age, city = get_person_details() #return 3
variables

# Displaying the results
print("Name:", name)
print("Age:", age)
print("City:", city)
```

```
ValueError: too many values to unpack (expected 3)
```

```
# Function returning multiple values
def get_person_details():
    name = "John"
    age = 30
    city = "Montréal"
```

```
    #gender = "Male"
    return name, age, city

# Unpacking with three variables
name, age, city = get_person_details() #3 return variables

# Displaying the results
print("Name:", name)
print("Age:", age)
print("City:", city)
```

```
Name: John
Age: 30
City: Montréal
```

Prefer raising exceptions to returning None

```
# Function that returns None on failure
def divide_numbers(a, b):
    if b == 0:
        return None  # Indicating failure by returning None
    else:
        return a / b

# Using the function and checking for failure with None
result = divide_numbers(10, 2)

if result is not None:
    print("Result:", result)
else:
    print("Error: Cannot divide by zero.")

# Using the function and checking for failure with None
result = divide_numbers(10, 0)

if result is not None:
    print("Result:", result)
else:
    print("Error: Cannot divide by zero.")
```

```
Result: 5.0
Error: Cannot divide by zero.
```

```python
# Function that raises an exception on failure
def divide_numbers(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    else:
        return a / b

# Using the function and handling the exception
try:
    result = divide_numbers(10, 2)
    print("Result:", result)
except ValueError as e:
    print("Error:", e)

# Using the function and handling the exception
try:
    result = divide_numbers(10, 0)
    print("Result:", result)
except ValueError as e:
    print("Error:", e)
```

```
Result: 5.0
Error: Cannot divide by zero
```

Know how Closures Interact with Variable Scope

- It is better to write a helper class compared to non-local or helper function.
- used specifically when we want to priortise certain groups in a function.

```python
class Sorter:
    def __init__(self, group):
        self.group = group
        self.found = False

    def __call__(self, x):
        if x in self.group:
            self.found = True
```

```
            return (0, x)
        else:
            return (1, x)

# Example usage
group = {2, 4, 6}
numbers = [5, 3, 2, 1, 4]

sorter = Sorter(group)
numbers.sort(key=sorter)

# Display the sorted list
print("Sorted List:", numbers)

# Check if any item from the group is found during sorting
assert sorter.found is True
```

```
Sorted List: [2, 4, 1, 3, 5]
```

Reduce Visual Noise with Variable Positional Arguments

*args is not suggested for two reasons-

```
1) Optional positional arguments are always turned into a tuple before they are passed to a f

2) Doesn't provide value inclusive of the new argument. Hence, no use of adding an additional
```

```
# Original function with *args
def example_function(*args):
    # Existing functionality using args
    total = sum(args)
    return total

# Example usage
result = example_function(1, 2, 3)
print("Result:", result)

# Attempt to add a new positional argument
# This would break existing callers
def updated_function(new_arg, *args):
```

```
    total = sum(args) + new_arg
    return total

result2 = updated_function(4,5)
print('Result2:', result2)
```

```
Result: 6
Result2: 9
```

Provide Optional Behavior with Keyword Arguments

```
def calculate_rectangle_area(length, width):
    return length * width
```

```
def calculate_rectangle_area(length, width=None):
    if width is not None:
        return length * width
    else:
        # If width is not provided, assume it's a square (width = length)
        return length * length
```

```
area1 = calculate_rectangle_area(5, 3)  # Calculates area of a rectangle
area2 = calculate_rectangle_area(4)     # Assumes it's a square with side length 4

print(area1)
print(area2)
```

```
15
16
```

Use None and Docstrings to Specify Dynamic Default Arguments

```
from datetime import datetime

def log_message(message, timestamp=None):
    """
    Log a message with an optional timestamp.
```

```
    Parameters:
    - message (str): The message to be logged.
    - timestamp (datetime, optional): The timestamp for the log message.
      Defaults to the current time if not provided.
    """
    if timestamp is None:
        timestamp = datetime.now()

    print(f"{timestamp}: {message}")

# Example usage
log_message("Error occurred")  # Logs the message with the current timestamp
log_message("Warning", timestamp=datetime(2023, 1, 1))  # Logs the message with a specific
```

```
2023-12-05 18:49:25.657917: Error occurred
2023-01-01 00:00:00: Warning
```

Define Function Decorators with funtools.wraps

- Decorator in Python is a function that takes another function as input and extends or modifies the behavior of the latter function.
- In this case, the trace decorator is designed to print information about the function calls.

```
def trace(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print(f'{func.__name__}({args!r}, {kwargs!r}) '
              f'-> {result!r}')
        return result
    return wrapper
```

```
@trace
def example_function(x, y):
    return x * y

result = example_function(3, 4)
```

```
example_function((3, 4), {}) -> 12
```

```
help(example_function)
```

```
Help on function wrapper in module __main__:

wrapper(*args, **kwargs)
```