Chapter 01 - Pythonic Thinking

May 8, 2021

0.1 Overview

- python programmers perfer to be explicit, to choose simple over complex and to maximize readability

[1]:
```
# find the version of python you are using

import sys
print(sys.version_info)
print(sys.version)
```

```
sys.version_info(major=3, minor=9, micro=2, releaselevel='final', serial=0)
3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)]
```

0.2 Item 2: Follow the PEP 8 Style Guide

0.2.1 Whitespace

- use spaces instead of tabs for indentation
- use four spaces for each level of syntactically significant indenting
- lines should be 79 chars in lenght
- continuation of long expressions into additional lines should be indented by four extra spaces from their normal indentation level
- functions and classes should be seprated by two blank lines
- methods should be seprated by one blank line
- in a dictionary, put no whitespace between each key and colon an put a single space before the corresponding value if it fits on the same line
- for type annotations, ensure that there is no sepration between the variable name and the colon and use a space before the type information

0.2.2 Naming

- functions, variables and attributes should be in `lowercase_underscore` format
- protected instance attributes should be in `_leading_underscore` format
- private instance attributes should be in `__double_leading_underscore_` format
- modlule-level constants should be in `ALL_CAPS`
- instance methods in classes should use `self`, which refers to the object, as the name of the first parameter
- class methods should use `cls` which refers to the class, as the name of the first parameter

0.2.3 Expressions and Statements

- use in line negation (`if a is not b`) instead of negation of postive expressions (`if not a is b`)
- dont check for empty containers or sequences (`like [] or "`) by comparing hte lenght to zero (`if len(somelist) == 0`). Use `if not somelist` and assume that empty values will implicitly evaluate to `False`
- the same thing goes for non-empty containers or sequences (`like [l] of 'hi'`). The statement if `somelist` is implicitly `True` for non-empty values
- avoid single-line `if` statements, `for` and `while` loops, and `except` compound statements. Spread these over multiple lines for clarity
- if you cant fit an expression on one line, surround it with parentheses and add line breaks and indentation to make it easier to read
- perfer surrounding multiline expressions with parentheses over using the `\` line contnuation character

0.3 Imports

- always put `import` statements (including `from x import y`) at the top of a file
- always use absolute names for modules when importing them, not relative to the current modules own path. For example, to import the `foo` module from within the `bar` package, you should use `from bar import foo` not just `import foo`
- if you must do relative imports, use the explicit syntax from `. import foo`
- imports should be in a section in the following order: `standard library modules`, `third-party modules`, `your own modules`. Each subsection should have imports in alphabetical order

0.4 Item 3: Know the Differences Between `bytes` and `str`

- in python there are two types that represent sequences of character data: `bytes` and `str`
- instances of `bytes` contain raw, unsigned 8-bit values (displayed in ASCII encoding

[2]:
```
a = b'h\x65llo'
print(list(a))
print(a)
```

```
[104, 101, 108, 108, 111]
b'hello'
```

- instances of `str` contain Unicode `code points` that represent textual characters from human language

[3]:
```
a = 'a\u0300 props'
print(list(a))
print(a)
```

```
['a', '̀', ' ', 'p', 'r', 'o', 'p', 's']
à props
```

- `str` instances do not have an associated binary encoding
  - call the `bytes()` method to convert the str

- `byte` instances do not have an assoicated text encoding
  - call the `str()` method to convert the bytes
- it is important to do encoding and decoding of unicode data at the furthest boundary of your interface
- the approach is called the `Unicode sandwich`
- the code of your program should use the `str` type containing Unicode data ans should not assume anything about character encoding

[5]:
```
# takes a byte or str instance and always returns str

def to_str(bytes_or_str):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode('utf-8')
    else:
        value = bytes_or_str
    return value

print(repr(to_str(b'foo')))
print(repr(to_str('bar')))
```

```
'foo'
'bar'
```

[8]:
```
# takes a byte or str instance and always returns a bytes
def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value

print(repr(to_bytes(b'foo')))
print(repr(to_bytes('bar')))
```

```
b'foo'
b'bar'
```

common gotchas: 1. `bytes` and `str` seem to work the same way but their instances are not compatible with each other

[14]:
```
# by using + operator, you can add bytes to bytes and str to str
print(b'one' + b'two')
print('one' + 'two')
print('')

# you cant add a byte to a str
# b'one' + 'two'
```

```
# - you can compare `bytes` to `str` by using binary operators
assert b'red' > b'blue'

# comparing bytes to str will result in false
print(b'foo' == 'foo')
print('')

# the % operator works with format strings for each type, respectively
print(b'red %s' % b'blue')
print('red %s' % 'blue')
print('')

# you can also pass a bytes instance to a str format strig
# the code invokes the __repr__ method
print('red %s' % b'blue')
```

```
b'onetwo'
onetwo

False

b'red blue'
red blue
red b'blue'
```

2. second issue is that operations involving file handles (returned by the `open` built-in function) defaults to requiring Unicode string instead of raw `bytes`

- a problem could be you open a file in write `w` mode and not binary write `wb` mode
- simmilarly you need to use `rb` and not `r` if expecting bytes
- you wnat to explicitly pass the `encoding` parameter

0.5 Item 4: Perfer Interpolated `F-String` over C-style format Strings and `str.format`

- the most common is to use `%`, which uses `C` style formatting
- you can escapte the formatting by using a double `%%`

[23]:
```
a = 0b10111011
b = 0xc5f
print('Binary is %d, hex is %d' % (a, b))
```

```
Binary is 187, hex is 3167
```

- python has the ability to also do formatting with a dictionary instead of a `tuple`
- the `keys` from the dictionary are matched with format specifiers with the corresponding name, such as `%(key)s`
- this approachs problem is that it makes code increase in `verbosity`
- each key must be specified at least twice

[24]:
```
key = 'my_var'
value = 1.234

new_way = '%(key)-10s = %(value).2f' % {
 'key': key, 'value': value}

print(new_way)
```

```
my_var     = 1.23
```

0.5.1 The `format` and `str.format`

- formatting behavior is specified by the `__format__` special method
- you can escape the str.format by using double `{{}}`

[25]:
```
a = 1234.5678
formatted = format(a, ',.2f')

print(formatted)
b = 'my string'

formatted = format(b, '^20s')
print('*', formatted, '*')
print()

# you can specify a placeholder
key = 'my_var'
value = 1.234

formatted = '{} = {}'.format(key, value)
print(formatted)

# way to think about how this works is that the format specifiers will be
# passed to the format function along with the vlaue format(value, '.2f')
formatted = '{:<10} = {:.2f}'.format(key, value)
print(formatted)
print("")

# you can also re-order the formatting
formatted = '{1} = {0}'.format(key, value)
print(formatted)
print("")

# the same positional index may also be referenced multiple times
formatted = '{0} loves food. See {0} cook.'.format(name)
print(formatted)
print("")
```

```
1,234.57
```

```
*      my string       *

my_var = 1.234
my_var     = 1.23

1.234 = my_var
```

- problem with `str.format` is that it is sill hard to read

0.5.2 Interpolated Format Strings

[27]:
```
key = 'my_var'
value = 1.234
formatted = f'{key} = {value}'
print(formatted)
print("")

formatted = f'{key!r:<10} = {value:.2f}'
print(formatted)
```

```
my_var = 1.234

'my_var'   = 1.23
```

[28]:
```
f_string = f'{key:<10} = {value:.2f}'

c_tuple = '%-10s = %.2f' % (key, value)

str_args = '{:<10} = {:.2f}'.format(key, value)

str_kw = '{key:<10} = {value:.2f}'.format(key=key,
 value=value)

c_dict = '%(key)-10s = %(value).2f' % {'key': key,
 'value': value}

assert c_tuple == c_dict == f_string
assert str_args == str_kw == f_string
```

0.6 Item 5: Write Helper Functions Instead of Complex Expressions

- unrelated, but this is a cool expression
  - `red = my_values.get('red',[""]) or 0`
- when you hace complex expressions, its better to add the logic to two variables
- if you are going to repeat code more than once use a helper function

0.7 Item 6: Perfer Multiple Assignment Unpacking over Indexing

- python has built-in tuple type that can be used to create immutable, ordered sequences of values
- a `tuple` is a pair of two values such as keys and values from a dictionary
- onces a tuple is created, you cannot modify it

[33]:
```
snack_calories = {
    'chips': 140,
    'popcorn': 80,
    'nuts': 190
}

items = tuple(snack_calories.items())
print(items)
```

```
(('chips', 140), ('popcorn', 80), ('nuts', 190))
```

[34]:
```
# values in a tuple can be accessed though numerical indexes
item = ('Peanut butter', 'Jelly')
first = item[0]
second = item[1]
print(first, 'and', second)
```

```
Peanut butter and Jelly
```

- python also has syntax for `unpacking` which allows for assigning multiple values in a single statement

[35]:
```
item = ('Peanut butter', 'Jelly')
first, second = item # Unpacking
print(first, 'and', second)
```

```
Peanut butter and Jelly
```

- unpacking has less visual noise than accessing the tuple's indexes and often requires fewer lines

[38]:
```
# lol you can actually unpack dictionaries

favorite_snacks = {
 'salty': ('pretzels', 100),
 'sweet': ('cookies', 180),
 'veggie': ('carrots', 20),
}
((type1, (name1, cals1)),
 (type2, (name2, cals2)),
 (type3, (name3, cals3))) = favorite_snacks.items()

print(f'Favorite {type1} is {name1} with {cals1} calories')
```

```
print(f'Favorite {type2} is {name2} with {cals2} calories')
print(f'Favorite {type3} is {name3} with {cals3} calories')
```

```
Favorite salty is pretzels with 100 calories
Favorite sweet is cookies with 180 calories
Favorite veggie is carrots with 20 calories
```

- unpacking can be used to swap values in place without the need to create temporary variables

[39]:
```
def bubble_sort(a):
    for _ in range(len(a)):
        for i in range(1, len(a)):
            if a[i] < a[i-1]:
                a[i-1], a[i] = a[i], a[i-1] # Swap

names = ['pretzels', 'carrots', 'arugula', 'bacon']
bubble_sort(names)
print(names)
```

```
['arugula', 'bacon', 'carrots', 'pretzels']
```

- unpacking can be using with enumerate

[40]:
```
snacks = [('bacon', 350), ('donut', 240), ('muffin', 190)]

for rank, (name, calories) in enumerate(snacks, 1):
    print(f'#{rank}: {name} has {calories} calories')
```

```
#1: bacon has 350 calories
#2: donut has 240 calories
#3: muffin has 190 calories
```

0.8 Item 7: Perfer `enumerate` over `range`

- `range` is useful for built-in function is useful for loops that iterate over a set of integers
- when you have a data structure to iterate over, like a list of strings, you cna loop directly over the sequences

[41]:
```
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
```

- often you want to iterate over a `list` and also know the index of the current item in the `list`
- using `range` to do this can make thing cluncky
- that is whey we have `enumerate`

[43]:
```
flavor_list = ['vanilla', 'chocolate', 'pecan', 'strawberry']

for i, flavor in enumerate(flavor_list, 0):
    print(f'{i + 1}: {flavor}')
```

```
1: vanilla
2: chocolate
3: pecan
4: strawberry
```

0.9 Item 8: Use `Zip` to process Iterators in Parallel

- sometimes you may have many lists of related objects such as a list of `names` and their `age`
- the items in the derived `list` are lated to the items in the source `list` by their indexes
- you could iterate over those lists by using `range` or `enumerate` but that is alot of noise
- thus we want to use `zip`

[47]:
```
names = ['Cecilia', 'Lise', 'Marie']
counts = [len(n) for n in names]

longest_name = None
max_count = 0

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

- `zip` wraps two or more iterators with a lazy generator
- the `zip` generator yeilds tuples containing the next value for each iterator
- these tuples can be unpacked directly within a `for` statement
- `zip` consumes the iterators it wraps one item at a time. which means it can be used with infinitely long inputs without risk of a program using too much memory and crashing
- warning with `zip` is that if you have lists with different sizes, `zip` will ignore some items if it does not have a pair
- if you dont expect lists of same behavior to be passed, consider using `zip_longest` from the `itertools` module

0.10 Item 9: Avoid `else` Blocks After `for` and `while` Loops

[48]:
```
for i in range(3):
    print('Loop', i)
else:
    print('Else block!')
```

```
Loop 0
Loop 1
```

```
Loop 2
Else block!
```

0.11 Item 10: Prevent Repetition with Assignment Expressions

- an assignment expression or a `walrus operator` is a new syntax introduced to solve a long-standing problem with the language that can cause code duplication
- the assignment is `a := b`
- walrus operator is useful because they enable you to assign variables in places where assignment statements are disallowed such as in the conditional expression of an if statement
- in the example below, we only use count once, but we need to define it above

[51]:
```
fresh_fruit = {
    'apple': 10,
    'banana': 8,
    'lemon': 5,
}

def make_lemonade(count):
    pass

def out_of_stock():
    pass
count = fresh_fruit.get('lemon', 0)
if count:
    make_lemonade(count)
else:
    out_of_stock()
```

- the pattern of fetching a value, checking to see if its non-zero, and then using it is extremely common

[55]:
```
if (count := fresh_fruit.get('lemon', 0)) >= 4:
        make_lemonade(count)
else:
    out_of_stock()
```

- Another common variation of this repetitve pattern occurs when I need to assign a variable in the enclosing scope depending on some condition, and then refrence that variable shortly afterward in a function call

[56]:
```
def slice_bananas(count):
    pass

class OutOfBananas(Exception):
    pass

def make_smoothies(count):
```

```
    pass

pieces = 0
count = fresh_fruit.get('banana', 0)
if count >= 2:
    pieces = slice_bananas(count)

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

- the walrus operator can again be used to shorten this example by one line of code
- this small change removes any emphasis on the count variable

[58]:
```
pieces = 0
if (count := fresh_fruit.get('banana', 0)) >= 2:
    prieces = slice_bananas(count)

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

- walrus operator also helps us with multiple ugle statements

[60]:
```
if (count := fresh_fruit.get('banana', 0)) >= 2:
    pieces = slice_bananas(count)
    to_enjoy = make_smoothies(pieces)
elif (count := fresh_fruit.get('apple', 0)) >= 4:
    to_enjoy = make_cider(count)
elif count := fresh_fruit.get('lemon', 0):
    to_enjoy = make_lemonade(count)
else:
    to_enjoy = 'Nothing'
```

- python also does not have a `do/while` loop
- the code below is repetitive because it requires two seprate `fresh_fruit` = 'pick_fruit() calls
- one before the loop to set initial conditions and another at the end of the loop to replensish the list of delivered fruits

[62]:
```
def pick_fruit():
    pass

def make_juice(fruit, count):
    pass

bottles = []
fresh_fruit = pick_fruit()
```

```
while fresh_fruit:
    for fruit, count in fresh_fruit.items():
        batch = make_juice(fruit, count)
        bottles.extend(batch)
    fresh_fruit = pick_fruit()
```

- the walrus operator inporves the code above

[63]:
```
bottles = []
while fresh_fruit := pick_fruit():
        for fruit, count in fresh_fruit.items():
            batck = make_juice(fruit, count)
            bottles.extend(batch)
```

Overview of Walrus: - basically if you just need to initalize a variable before using it in an `if` statement, use walrus - if you want to emulate the `switch/case` or `do/while` use the walrus