

13 Tip

Clean code

Variables

Functions



Targets Python3.7+



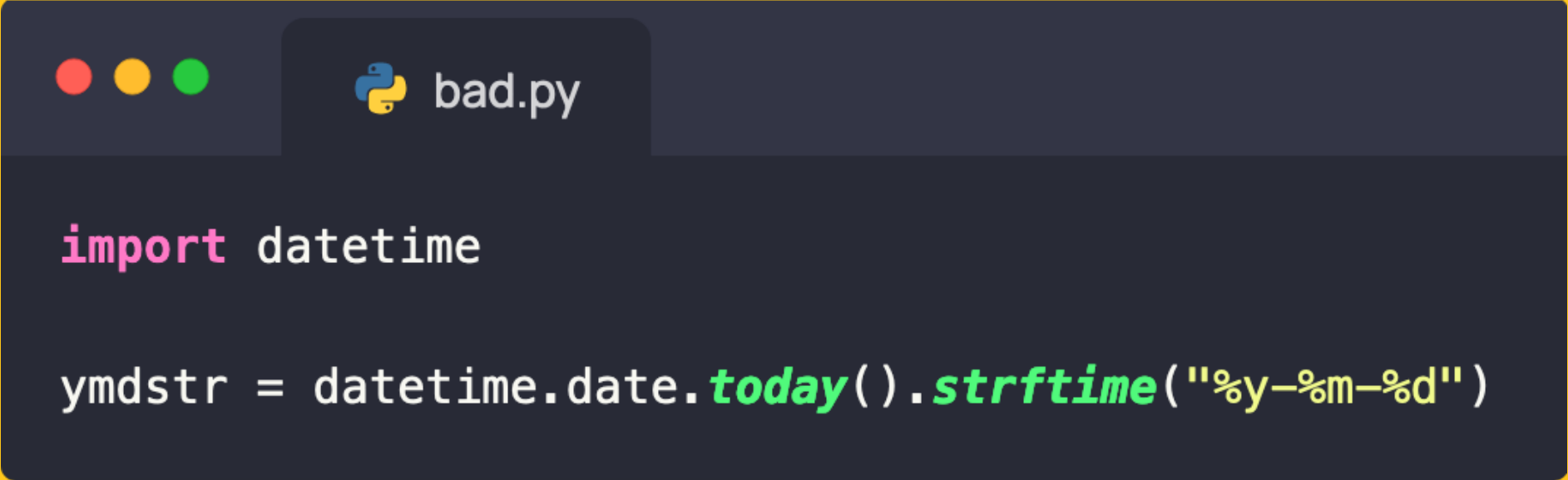
@miladkoochi



1

Variables

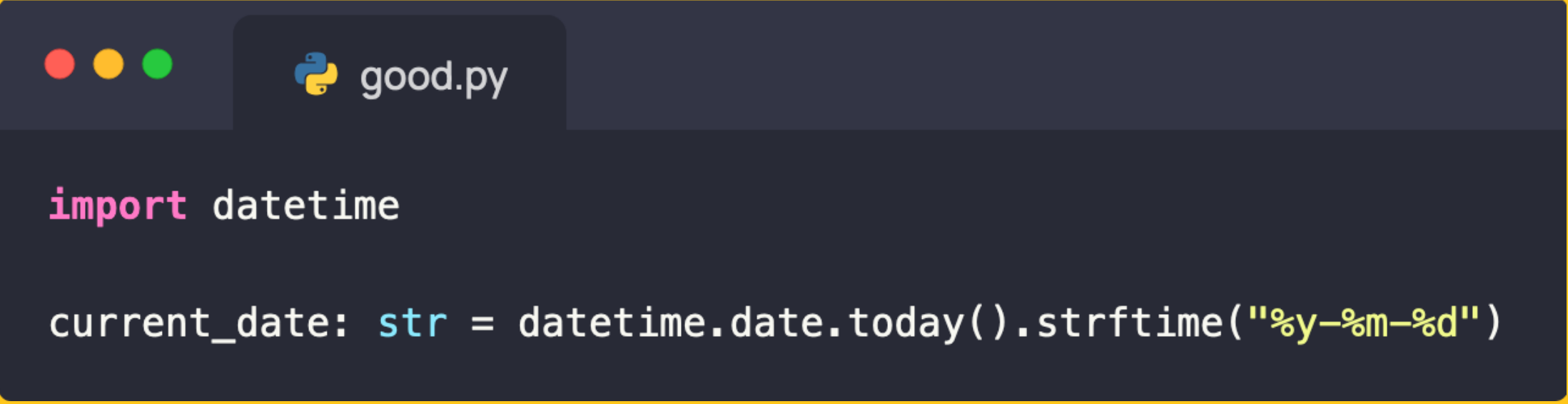
Use meaningful and pronounceable variable names



```
import datetime

ymdstr = datetime.date.today().strftime("%y-%m-%d")
```

Additionally, there's no need to add the type of the variable (str) to its name.



```
import datetime

current_date: str = datetime.date.today().strftime("%y-%m-%d")
```

2

Variables

Use the same vocabulary for the same type of variable

```
def get_user_info(): pass

def get_client_data(): pass

def get_customer_record(): pass
```

Good: If the entity is the same, you should be consistent in referring to it in your functions:

```
def get_user_info(): pass

def get_user_data(): pass

def get_user_record(): pass
```

Even better:



2

Variables

Even better Python is (also) an object oriented programming language. If it makes sense, package the functions together with the concrete implementation of the entity in your code, as instance attributes, property methods, or methods:

```
from typing import Union, Dict

class Record:
    pass

class User:
    info: str

    @property
    def data(self) -> Dict[str, str]:
        return {}

    def get_record(self) -> Union[Record, None]:
        return Record()
```

3

Variables

Use searchable names

We will read more code than we will ever write. It's important that the code we do write is readable and searchable. By not naming variables that end up being meaningful for understanding our program, we hurt our readers. Make your names searchable.

```
import time
```

```
# What is the number 86400 for again?  
time.sleep(86400)
```

```
import time
```

```
# Declare them in the global namespace for the module.  
SECONDS_IN_A_DAY = 60 * 60 * 24  
time.sleep(SECONDS_IN_A_DAY)
```

4

Variables

Use explanatory variables

```
import re

address = "One Infinite Loop, Cupertino 95014"
city_zip_code_regex = r"^[^,\s]+[,,\s]+(.*?)\s*(\d{5})?$"
matches = re.match(city_zip_code_regex, address)
if matches:
    print(f"{matches[1]}: {matches[2]}")
```

Good:

Decrease dependence on regex by naming subpatterns.

```
import re

address = "One Infinite Loop, Cupertino 95014"
city_zip_code_regex = r"^[^,\s]+[,,\s]+(?P<city>.*?)\s*(?P<zip_code>\d{5})?$"

matches = re.match(city_zip_code_regex, address)
if matches:
    print(f"{matches['city']}, {matches['zip_code']}")
```

5

Variables

Avoid Mental Mapping

```
seq = ("Austin", "New York", "San Francisco")

for item in seq:
    # do_stuff()
    # do_some_other_stuff()

    # Wait, what's `item` again?
    print(item)
```

Don't force the reader of your code to translate what the variable means. Explicit is better than implicit.

```
locations = ("Austin", "New York", "San Francisco")

for location in locations:
    # do_stuff()
    # do_some_other_stuff()
    # ...
    print(location)
```

6

Variables

Don't add unneeded context



bad.py

```
class Car:  
    car_make: str  
    car_model: str  
    car_color: str
```

If your class/object name tells you something, don't repeat that in your variable name.



good.py

```
class Car:  
    make: str  
    model: str  
    color: str
```




Variables

Use default arguments instead of short circuiting or conditionals

```
import hashlib

def create_micro_brewery(name):
    name = "Hipster Brew Co." if name is None else name
    slug = hashlib.sha1(name.encode()).hexdigest()
    # etc.
```

... when you can specify a default argument instead? This also makes it clear that you are expecting a string as the argument.

```
import hashlib

def create_micro_brewery(name: str = "Hipster Brew Co."):
    slug = hashlib.sha1(name.encode()).hexdigest()
    # etc.
```

8

Functions

Functions should do one thing

```
bad.py

from typing import List

class Client:
    active: bool

def email(client: Client) -> None:
    pass

def email_clients(clients: List[Client]) -> None:
    for client in clients:
        if client.active:
            email(client)
```

```
good.py

from typing import Generator, Iterator

class Client:
    active: bool

def email(client: Client):
    pass

def active_clients(clients: Iterator[Client]) -> Generator[Client, None, None]:
    return (client for client in clients if client.active)

def email_client(clients: Iterator[Client]) -> None:
    for client in active_clients(clients):
        email(client)
```

9

Functions

Function arguments (2 or fewer ideally)

A large amount of parameters is usually the sign that a function is doing too much (has more than one responsibility). Try to decompose it into smaller functions having a reduced set of parameters, ideally less than three.

If the function has a single responsibility, consider if you can bundle some or all parameters into a specialized object that will be passed as an argument to the function. These parameters might be attributes of a single entity that you can represent with a dedicated data structure. You may also be able to reuse this entity elsewhere in your program. The reason why this is a better arrangement is than having multiple parameters is that we may be able to move some computations, done with those parameters inside the function, into methods belonging to the new object, therefore reducing the complexity of the function.



9

Functions

Function arguments (2 or fewer ideally)

```
def create_menu(title, body, button_text, cancellable):  
    pass
```

```
class MenuConfig:  
    title: str  
    body: str  
    button_text: str  
    cancellable: bool = False  
  
def create_menu(config: MenuConfig) -> None:  
    title = config.title  
    body = config.body  
    # ...  
  
config = MenuConfig()  
config.title = "My delicious menu"  
config.body = "A description of the various items on the menu"  
config.button_text = "Order now!"  
# The instance attribute overrides the default class attribute.  
config.cancellable = True  
  
create_menu(config)
```



9

Functions

Function arguments (2 or fewer ideally)

```
from dataclasses import astuple, dataclass

@dataclass
class MenuConfig:

    title: str
    body: str
    button_text: str
    cancellable: bool = False

def create_menu(config: MenuConfig):
    title, body, button_text, cancellable = astuple(config)
    # ...

create_menu(
    MenuConfig(
        title="My delicious menu",
        body="A description of the various items on the menu",
        button_text="Order now!"
    )
)
```



9

Functions

Function arguments (2 or fewer ideally)

+Even fancier, Python3.8+ only

```
from typing import TypedDict

class MenuConfig(TypedDict):

    title: str
    body: str
    button_text: str
    cancellable: bool

def create_menu(config: MenuConfig):
    title = config["title"]
    # ...

create_menu(
    # You need to supply all the parameters
    MenuConfig(
        title="My delicious menu",
        body="A description of the various items on the menu",
        button_text="Order now!",
        cancellable=True
    )
)
```

10

Functions

Function names should say what they do



bad.py

```
class Email:
    def handle(self) -> None:
        pass

message = Email()
# What is this supposed to do again?
message.handle()
```



good.py

```
class Email:
    def send(self) -> None:
        """Send this message"""

message = Email()
message.send()
```

11

Functions

Functions should only be one level of abstraction

When you have more than one level of abstraction, your function is usually doing too much. Splitting up functions leads to reusability and easier testing.

```
def parse_better_js_alternative(code: str) -> None:
    regexes = [
        # ...
    ]
    statements = code.split('\n')
    tokens = []
    for regex in regexes:
        for statement in statements:
            pass
    ast = []
    for token in tokens:
        pass
    for node in ast:
        pass
```



11

Functions

Functions should only be one level of abstraction

```
from typing import Tuple, List, Dict

REGEXES: Tuple = (
    # ...
)

def parse_better_js_alternative(code: str) -> None:
    tokens: List = tokenize(code)
    syntax_tree: List = parse(tokens)

    for node in syntax_tree:
        pass

def tokenize(code: str) -> List:
    statements = code.split()
    tokens: List[Dict] = []
    for regex in REGEXES:
        for statement in statements:
            pass

    return tokens

def parse(tokens: List) -> List:
    syntax_tree: List[Dict] = []
    for token in tokens:
        pass
    return syntax_tree
```

12

Functions

Don't use flags as function parameters



```
from tempfile import gettempdir
from pathlib import Path

def create_file(name: str, temp: bool) -> None:
    if temp:
        (Path(gettempdir()) / name).touch()
    else:
        Path(name).touch()
```



```
from tempfile import gettempdir
from pathlib import Path

def create_file(name: str) -> None:
    Path(name).touch()

def create_temp_file(name: str) -> None:
    (Path(gettempdir()) / name).touch()
```

13

Functions

Avoid side effects

A function produces a side effect if it does anything other than take a value in and return another value or values. For example, a side effect could be writing to a file, modifying some global variable, or accidentally wiring all your money to a stranger.

Now, you do need to have side effects in a program on occasion - for example, like in the previous example, you might need to write to a file. In these cases, you should centralize and indicate where you are incorporating side effects. Don't have several functions and classes that write to a particular file - rather, have one (and only one) service that does it.

The main point is to avoid common pitfalls like sharing state between objects without any structure, using mutable data types that can be written to by anything, or using an instance of a class, and not centralizing where your side effects occur. If you can do this, you will be happier than the vast majority of other programmers.



13

Functions

Avoid side effects

```
bad.py

fullname = "Ryan McDermott"

def split_into_first_and_last_name() -> None:
    global fullname
    fullname = fullname.split()

split_into_first_and_last_name()

# MyPy will spot the problem, complaining about 'Incompatible types in
# assignment: (expression has type "List[str]", variable has type "str")'
print(fullname) # ["Ryan", "McDermott"]
```

```
good.py

from dataclasses import dataclass

@dataclass
class Person:
    name: str

    @property
    def name_as_first_and_last(self) -> list:
        return self.name.split()

# The reason why we create instances of classes is to manage state!
person = Person("Ryan McDermott")
print(person.name) # => "Ryan McDermott"
print(person.name_as_first_and_last) # => ["Ryan", "McDermott"]
```



im milad koohi

Senior Software Developer



@miladkoohi

if like repost

