#readings #python #topublish

Effective Python

See also the GitHub page.

Chapter 3: Functions

19: Never unpack more than three variables when functions return multiple values

For two reasons: 1. Easy to use the wrong order of the return values, thereby introducing bugs; 2. Hurts PEP8 style;

Instead of this, return a small class or a `namedtuple` instance.

20: Prefer raising exceptions to returning `None`

- Functions that return `None` to indicate special meaning are error prone because `None` and other values (zero, empty string, …) all evaluate to `False` in conditional expressions.
- *Never return `None` for special cases.* Instead, raise an `Exception` up to the caller and let it deal with it. Expect this to handle exceptions properly when these are documented.
- Use type annotations to make it clear that a function will not return `None`.

21: Know how closures interact with variable scope

A common way to prioritize elements is by passing an helper function as the `key` argument to a list's `sort` method (see *14*). The helper can check whether the given item is in the important group and can vary the sorting value accordingly.

```
def sort_priority(values, group):
    def helper(x):
        if x in group:
            return (0, x)
        return (1, x)
    values.sort(key = helper)
numbers = [8,3,1,2,5,4,7,6]
group = {2,3,5,7}
sort_priority(numbers, group)
```

There are 3 reasons why this function operates as expected:

1. Python supports *closures* - functions that refer to variables from the scope in which they were defined. *In this particular case, `helper` is defined within `sort_priority`, and therefore one can use `group` inside `helper`*;
2. Functions are *first-class* objects in Python, which means that one can refer to them directly, assign them to variables, pass them as arguments to other functions, compare them in expressions and `if` statements, … ;
3. Python has specific rules for comparing sequences (see page 107);

In Python, the `nonlocal` keyword can be used for getting data out of a closure. Note, however, that a `nonlocal` variable will not transverse to module-level scope: this is done to avoid polluting global variables.

This should, however, be avoided. Furthermore, when the usage of `nonlocal` starts getting complicated, it is better to wrap your state in a helper class.

```
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)
        return (1,x)
```

22: Reduce visual noise with variable positional arguments

Two problems (with accepting a variable number of positional arguments):

1. Optional positional arguments (`*args`) are always turned into a tuple before they are passed to a function. This means that if the caller of a function uses the `*` operator on a generator, it will be iterated until it is exhausted, which could consume a lot of memory and cause the program to crash. Therefore, it is better to use this pattern when the number of values is small;
2. One cannot add new positional arguments without migrating every caller, which can introduce bugs. To avoid this possibility, you should use keyword-only arguments when you want to extend functions that accept `*args`;

23: Provide optional behavior with keyword arguments

The keyword arguments can be passed in any order as long as all of the required positional arguments are specified. That is, for the function

```
def remainder(number, divisor):
    return number % divisor
```

the calls

```
remainder(number=20, divisor=5)
remainder(divisor=5, number=20)
```

are identical. Note, however, that positional arguments need to be specified before keyword arguments, otherwise a `SyntaxError` is generated. If the input values are part of a dictionary, `my_kwargs = {"number": 20, "divisor": 5}`, you can pass these as keyword arguments of a function, `remainder(**my_kwargs)`.

Three benefits of keyword arguments:

1. Clarity of what the inputs are;
2. Allowing default values, thereby reducing duplicate code;

3. Extend a function's parameters while remaining backwards compatible;

The best practice is to always specify optional arguments using the keyword names and never pass them as positional arguments.

24: Use `None` and Docstrings to specify dynamic default arguments

A default argument value is evaluated *once* per module load, which usually happens when the program starts up. The convention for achieving a variable default argument (see example in pages 117-118) is to provide a default value of `None` and to document the actual behavior in the docstring. When the code sees the argument value `None`, it allocates the default value accordingly.

```
from datetime import datetime

def log(message, when=None):
    if when is None:
        when = datetime.now()
    print(f'{when}: {message}')
```

25: Enforce clarity with Keyword-Only and Positional-Only arguments

The `*` symbol in the argument list indicates the end of the positional arguments and the beginning of keyword-only arguments:

```
def safe_division(number,
divisor,
*,
ignore_overflow=False,
ignore_zero_division=False)
```

The `/` symbol indicates where positional-*only* arguments end:

```
def safe_division(number,
divisor,
/,
*,
ignore_overflow=False,
ignore_zero_division=False)
```

One notable consequence of keyword and positional-only arguments is that any parameter name between the `/` and `*` symbols in the argument list can either be passed by position OR by keyword (default in Python).

26: Define Function Decorators with `functools.wraps`

Because a decorator can run additional code before and after the function it wraps, it can access and modify input arguments, return values and raise exceptions. However, if you simply define a decorator, you lose functionality, like `print`, `help` or even pickling.

Therefore, it is better to use `from functools import wraps` and use this a decorator in the `wrapper` functionality inside the decorator. As a result, one is able

to copy all the important metadata about the inner function (e.g., `fibonacci`) to the outer function.

Chapter 4: Comprehensions and Generators

27: Use comprehensions instead of `map` and `filter`

28: Avoid more than two control subexpressions in comprehensions

Comprehensions support multiple `if` conditions. Multiple conditions at the same loop level have an implicit `and`. Avoid, however, using comprehensions which contain conditions at multiple levels.

Avoid using more than two control subexpressions in a comprehension. More than this, use `ifs` in a helper function. In other words, two conditions, two loops or one condition and one loop.

29: Avoid repeated work in comprehensions by using assignment expressions

To avoid writing `{name: get_batches(stock.get(name,0), 8) for name in order if get_batches(stock.get(name,0), 8)}`, you can instead use the walrus `:=` operator, which forms an *assignment expression*:

```
{name: batches for name in order
if (batches := get_batches(stock.get(name,0), 8)})
```

Here the walrus operator is being used in the conditional part.

Note, however, that if a comprehension uses the walrus operator in the value part of the comprehension *and does not have a condition*, it will leak the loop variable to the containing scope. As a result, *use assignment expressions only on the condition part of the comprehension.*

30: Consider generators instead of returning lists

```
def index_words(text):
    result = []
    if text:
        result.append(0)
    for index, letter in enumerate(text):
        if letter == ' ':
            result.append(index + 1)
    return result
```

Instead, this is cleaner:

```
def index_words_iter(text):
    if text:
        yield 0
    for index, letter in enumerate(text):
        if letter == ' ':
            yield index + 1
```

When called, a generator function does not actually run but instead returns an iterator. With each call to the `next` built-in function, the iterator advances the generator to its next yield expression.

The one drawback of defining generators is that the iterators returned are stateful and therefore cannot be reused.

31: Be defensive when iterating over arguments

`for` loops and `lists` expect the `StopIteration` exception to be raised during normal operation. As a result, these cannot tell the difference between an iterator that has no output and an iterator that had an output but is now exhausted.

To solve this problem, one can completely exhaust an input iterator and keep a copy of its entire contents in a `list`. *However, this would fail when the output of the iterator function is very large.* One way to get around this is to accept a function that returns a new iterator each time it is called by using `get_iter()`, if `get_iter` is the input iterator. To then use the function which accepts an iterator, we can pass a `lambda` function (see page 142).

Because this is rather clumsy, it is better to use a container class that implements the `iterator` protocol. The iterator protocol is how `for` loops and related expressions traverse the contents of a container type.

When Python sees a statement like `for x in foo`, it actually calls `iter(foo)`. The `iter` built-in function then calls `foo.__iter__` special method in turn. This special method must return an iterator object. Then the `for` loop repeatedly calls the `next` built-in function on the iterator object until it is exhausted, as indicated by raising a `StopIteration` exception.

For example:

```
class ReadVisits:
    def __init__(self, data_path):
        self.data_path = data_path

    def __iter__(self):
        with open(self.data_path) as f:
            for line in f:
                yield int(line)
```

The new container type works correctly when passed to the original function, `normalize`. This is because the `sum` method calls `ReadVisits.__iter__` to allocate a new iterator object. The subsequent `for` loop to normalize the number also calls `ReadVisits.__iter__` to allocate a second iterator object.

One downside of this approach is that the data has to be read multiple times.

One can also program more defensively by checking if the input of `normalize` is an iterator or a container. If it is an iterator, we want to raise a `TypeError` .

32: Consider Generator Expressions for Large List Comprehensions

When the size of the input data in a list comprehension becomes a problem, use a generator expression instead. These do not materialize the whole output sequence when they are run. Instead, they evaluate to an iterator that yields one item at a time from the expression. Note that these can be chained together.

33: Compose multiple generators with `yield from`

34: Avoid injecting data into generators with `send`

When the `send` method is called instead of iterating the generator with a `for` loop or the `next` built-in function, the supplied parameter becomes the value of the `yield` expression when the generator is resumed. The problem with using `send` is that it does not work as expected when multiple `yield from` are being called.

35: Avoid causing state transitions in generators with `throw`

When `throw` is called, the next occurrence of a `yield` expression re-raises the provided `Exception` instance after its output is received. However, it is best to avoid `throw` and instead use an iterable class if you need this (e.g., mixing generators and exceptions) type of behavior.

36: Consider `itertools` for working with iterators and generators

Whenever you find yourself dealing with tricky iteration code, it is worth looking at the `itertools` documentation.

Linking iterators together `chain` , `repeat`, `cycle`, `tee`, `zip_longest`

Filtering items from an iterators `islice`, `takewhile`, `dropwhile`, `filterfalse`

Producing combinations of items from iterators `accumulate`, `product` (returns the Cartesian product of items from one or more iterators), `permutations`, `combinations`, `combinations_with_replacement`

Chapter 5: Classes and Interfaces

As an OO programming language, Python supports inheritances, polymorphism and encapsulation.

37: Compose Classes Instead of Nesting Many Levels of Built-in Types

Python's dictionary type is wonderful for maintaining dynamical internal stat information over the lifetime of the object. However, there is the danger of overextending them to write brittle code. As soon as your bookkeeping gets complicated, break it all out into classes.

- Provide well-defined interfaces that better encapsulate your data.

- Enables you to create a layer of abstraction between your interfaces and your concrete implementations.

Think also which constructs go along together.

The pattern of extending tuples longer and longer is similar to deepening layers of dictionaries. As soon as you find yourself going longer than a two-tuple, it is time to consider another approach (for example, `namedtuple`). This allows one to define tiny, immutable data classes.

See example pages 171-174.

The hierarchy of classes is:

`Grade` -> `Subject` -> `Student` -> `Gradebook`

`Grade` is the fundamental object (here a `namedtuple`), a `Subject` is composed of multiple `Grades`, a `Student` is composed of multiple subjects, a `Gradebook` is composed of multiple `Students`.

38: Accept Functions Instead of Classes for Simple Interfaces

Functions work as hooks because Python as first-class functions: these and *methods* can be passed around and referenced like any other value in the language.

One example where a class method is used as a stateful hook:

```
class BetterCountMissing:
    def __init__(self):
        self.added = 0
    def __call__(self): # if we had used missing,
        # we would have to specify counter.missing below
        self.added += 1
        return 0

from collections import defaultdict
current = {'green': 12, 'blue': 3}
counter = BetterCountMissing() # instance of BetterCountMissing class,
# which when called adds 1 to added and returns zero
result = defaultdict(counter, current)

for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

The call method indicates that a class's instances will be used somewhere a function argument could also be suitable.

- *The `__call__` special method enables instances of a class to be called like plain Python functions.*
- When you need a function to maintain state, consider defining a class that provides the `__call__` method instead of defining a stateful closure.

39: Use Polymorphism to construct objects generically

In Python, both functions and classes support polymorphism. This enables multiple classes in a hierarchy to implement their own unique versions of a method. As a result, several classes can fulfill the same interface or abstract base class while providing different functionality.

To have a generic way to construct objects, each `InputData` class should provide a special constructor that can be used generically by the helper methods that orchestrate MapReduce (see original below). *The problem is that Python only allows for a single constructor method* `__init__`. The best way to solve this problem is with class method polymorphism: works similar to the instance method polymorphism - used for `InputData.read` - but for whole classes.

```
class GenericInputData:
    def read(self):
        raise NotImplementedError

    @classmethod
    def generate_inputs(cls, config):
        raise NotImplementedError
```

`generate_inputs` 's `config` is a dictionary with a set of configuration parameters that the `GenericInputData` concrete subclass needs to interpret. `PathInputData` implements `generic_inputs`:

```
class PathInputData(GenericInputData):
    ...

    @classmethod
    def generate_inputs(cls, config):
        data_dir = config['data_dir']
        for name in os.listdir(data_dir):
            yield clas(os.path.join(data_dir,name))
```

The `input_class` parameter, which must be a subclass of `GenericInputData`, is used to generate the necessary inputs in `GenericWorker`:

```
class GenericWorker:
    def __init__(self, input_data):
        self.input_data = input_data
        self.result = None

    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError

    @classmethod
    def create_workers(cls, input_class, config):
        workers = []
        for input_data in input_class.generate_inputs(config):
```

```
            workers.append(cls(input_data))
        return workers
```

`create_workers` calling `cls()` provides an alternative way to construct `GenericWorker` objects besides using the `__init__` method directly.

```
class LineCountWorker(GenericWorker):
    def map(self):
        data = self.input_data.read() # <--- This is where the
        # read method of PathInputData is being used
        self.result = data.count('\n')

    def reduce(self, other):
        self.reduct += other.result
```

The `mapreduce` function is then completely generic:

```
# original mapreduce

def mapreduce(data_dir):
    inputs = generate_inputs(data_dir)
    workers = create_workers(inputs)
    return execute(workers)

# new mapreduce

def mapreduce(workerclass, input_class, config):
    workers = worker_class.create_workers(input_class, config)
    return execute(workers)
```

Now one can write other `GenericInputData` and `GenericWorker` subclasses without having to rewrite any of the glue code.

- Python only supports a single constructor per class: `__init__`;
- Use `@classmethod` to define alternative constructors for your classes;
- Use class method polymorphism to provide generic ways to build and connect many concrete subclasses;

40: Initialize parent classes with `super`

The old way of initializing a parent class from the child class is to directly class the parent's `__init__` method with the child's instance:

```
class MyBaseClass:
    def __init__(self, value):
        self.value = value

class MyChildClass:
    def __init__(self):
        MyBaseClass.__init__(self,5)
```

This approach works fine for basic class hierarchies but breaks in many cases. *If a class is affected by multiple inheritance (something to avoid in general),*

calling the superclassses' `__init__` methods directly can lead to unpredictable behavior.

This is because the `__init__` call order is not specified across all subclasses:

1. When class ordering is ill-defined
2. When diamond inheritance (when a subclass inherits from two separate classes that have the same superclass somewhere in the hierarchy) exists

See page 185 for class ordering and diamond inheritance.

To solve these problems, Python has the `super` built-in function and standard method resolution order (MRO):

- The former ensures that common superclasses in diamond inheritances are run only once;
- The latter defines the ordering with which superclasses are initialized;

The ordering is not very obvious, see page 186.

```
class MyBaseClass:
    def __init__(self, value):
        self.value = value

class TimesSevenCorrect(MyBaseClass):
    def __init__(self, value):
        super().__init__(value)
        self.value *= 7

class PlusNineCorrect(MyBaseClass):
    def __init__(self, value):
        super().__init__(value)
        self.value += 9

class GoodWay(TimesSevenCorrect, PlusNineCorrect):
    def __init__(self, value):
        super().__init__(value)
```

The super function can also be called with two parameters: first the type of the class whose MRO parents view you are trying to access. Second, the instance on which to access that view. However, this is not necessary, and the only time you should provide parameters to `super` is in situations where you need to access the specific functionality of a superclass's function implementation from a child class.

41: Consider composing functionality with Mix-in classes

If you find yourself desiring the convenience and encapsulation that come with multiple inheritance, but want to avoid the potential headaches, consider writing a mix-in instead. The mix-in is a class that defines a small set of additional methods for its child classes to provide. *Mix-in classes do not define their own instance attributes nor require their `__init__` constructor to be called.*

Dynamic inspection means one can write generic functionality just once (in a mix-in) and then apply it to many other classes.

Example: convert a Python object from its in-memory representation to a dictionary that is ready for serialization.

```
class ToDictMixin:

     def to_dict(self):
         return self._transverse_dict(self.__dict__)

    def _transverse_dict(self, instance_dict):
        output = {}
        for key, value in instance_dict.items():
            output[key] = self._transverse(key, value)
        return output

    def _transverse(self, key, value):
        if isinstance(value, ToDictMixin):
            return value.to_dict()
        elif isinstance(value, dict):
            return self._transverse_dict(value)
        elif isinstance(value, list):
            return [self._transverse(key, i) for i in value]
        elif hasattr(value, '__dict__'):
            return self._transverse_dict(value.__dict__)
        else:
            return value
```

The best part about mix-ins is that you can make their generic functionality pluggable so behaviors can be overriden when required. See example page 190. These also can be composed together and use both class and instance methods.

42: Prefer public attributes over private ones

The only time to seriously consider using private attributes is when one is worried about naming conflicts with subclasses. This problem already occurs when a child class unwittingly defines an attribute that was already defined by its parent class. This is primarily a concern with classes that are part of a public API: the subclasses are out of your control, so refactor does not fix the problem.

To reduce the risk of this issue occurring, you can have a private attribute in the parent class to ensure that there are no attribute names that overlap with the child classes

43: Inherit from `collections.abc` for customer container types

Much of programming in Python is defining classes that contain data and describing how such objects relate to each other.

An example with `IndexableNode` (page 199) shows that to define a sequence from scratch requires custom implementation for several methods (`__getitem__`, `__len__`, `__count__` `__index__`). To avoid this difficulty, one

can use `collections.abc`, which defines a set of abstract base classes that provide all the typical methods for each container type.

When you implement all the methods required by an abstract base class (as done with `SequenceNode`), it provides all of the additional methods, like `index` and `count` for free. The benefit of using these abstract base classes is even greater for more complex container types like `Set` and `MutableMapping`, which have a large number of special methods that need to be implemented to match Python conventions.

Chapter 6: Metaclasses and Attributes

Metaclasses let you intercept Python's `class` statement and provide special behavior each time a class is defined. One can use Python's built-in features to dynamically customize attributes. However, both can lead to a lot of confusion and therefore should be used with caution.

44: Use plain attributes instead of setter and getter methods

Using setters and getters is simple but not Pythonic and are especially clumsy for certain operations, like incrementing in place. *These utility methods do, however, help define the interface for a class, making it easier to encapsulate functionality, validate usage and define boundaries.*

In Python, you never need to implement explicit setter or getter methods and instead start your implementation with public attributes (see example page 205).

If special behavior is necessary when an attribute is set, use the `@property` decorator and the corresponding `setter` attribute. Specifying a setter also allows one to perform type checking and validation on values passed to the class.

The best policy is to modify only related object state in `@property.setter` methods.

45: Consider `@property` instead of refactoring attributes

One advanced but common use of `@property` is transitioning what was once a simple numerical attribute into an on-the-fly computation. This is extremely helpful because it lets one migrate all existing usage of a class to have new behaviors without requiring any of the call sites to be rewritten.

```
from datetime import datetime, timedelta

class Bucket:
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.quota = 0

def fill(bucket, amount):
    now = datetime.now()
```

```
    if (now - bucket.reset_time) > bucket.period_delta:
        bucket.quota = 0
        bucket.reset_time = now
    bucket.quota += amount

def deduct(bucket, amount):
    now = datetime.now()
    if (now - bucket.reset_time) > bucket.period_delta:
        return False
    if bucket.quota - amount < 0:
        return False
    bucket.quota -= amount
    return True
```

See page 211: To match the previous interface of the original bucket class, use the `@property` method to compute the current level of quota on the fly through `self.max_quota` and `self.quota_consumed`.

```
from datetime import datetime, timedelta

class NewBucket:
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.max_quota = 0
        self.quota_consumed = 0

    @property
    def quota(self):
        return self.max_quota - self.quota_consumed

    @quota.setter
    def quota(self, amount):
        delta = self.max_quota - amount
        if amount == 0:
            self.max_quota = 0
            self.quota_consumed = 0
        elif delta < 0:
            assert self.quota_consumed == 0
            self.max_quota = amount
        else:
            assert self.max_quota >= self.quota_consumed
            self.quota_consumed += delta

def fill(bucket, amount):
    now = datetime.now()
    if (now - bucket.reset_time) > bucket.period_delta:
        bucket.quota = 0
        bucket.reset_time = now
    bucket.quota += amount
```

```
def deduct(bucket, amount):
    now = datetime.now()
    if (now - bucket.reset_time) > bucket.period_delta:
        return False
    if bucket.quota - amount < 0:
        return False
    bucket.quota -= amount
    return True
```

The best part is that the code using `Bucket.quota` does not have to change or know that the class has changed. New usage of `Bucket` can do the right thing and access `max_quota` and `quota_consumed` directly.

46: Use descriptors for reusable `@property` methods

The property built-in is not reusable. The methods it decorates cannot be reused for multiple attributes of the same class and they cannot also be reused by unrelated classes.

See page 213 and 214 for an example where the `@property` method leads to a lot of boilerplate code.

The better way to do this in Python is to use a *descriptor*. The descriptor protocol defines how attribute access is interpreted by the language. A descriptor class can provide `__get__` and `__set__` methods that let you re-use the same logic (in this case, grade validation) without boilerplate. For this purpose, they are better suited than mix-ins.

Define a new class called `Exam` with *class attributes* which are `Grade` instances. The `Grade` class implements the descriptor protocol:

```
class Grade:
    def __get__(self, instance, instance_type):
        ...

    def __set__(self, instance, value):
        ...

class Exam:
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()
```

```
exam = Exam()
exam.writing_grade = 40
```

is interpreted as `Exam.__dict__['writing_grade'].__set__(exam, 40)`, whereas

```
exam.writing_grade
```

is interpreted as `Exam.__dict__['writing_grade'].__get__(exam, Exam)`. What drives this behavior is `__getattribute__` method of `object`. When an `Exam` *instance* does not have an *attribute* named `writing_grade`, Python falls

back to the `Exam` *class*'s *attribute* instead. *If this class attribute is an object that has `__get__` and `__set__` methods, Python assumes that you want to follow the descriptor protocol.*

The problem with the implementation of `Grade` in page 215 is that *a single `Grade` instance is shared across all `Exam` instances for the class attribute `writing_grade`.* The `Grade` instance is constructed once in the programs lifetime, when the `Exam` class is first defined, not each time an `Exam` instance is created.

To solve this, one needs the `Grade` class to keep track of its value for each unique `Exam` instance.

```
class Grade:
    def __init__(self):
        self._values = {}

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return self._values.get(instance, 0)

    def __set__(self, instance, value):
        if not (0 <= value <= 100):
            raise ValueError(...)
        self._values[instance] = value
```

However, this implementation leaks memory. To fix this one can use `weakref`'s `WeakKeyDictionary`. The unique behavior of `WeakKeyDictionary` is that it removes `Exam` instances from its set of items when the Python runtime knows it is holding the instance's last remaining reference in the program.

```
from weakref import WeakKeyDictionary

class Grade:
    def __init__(self):
        self._values = WeakKeyDictionary()
```

47: Use `__getattr__`, `__getattribute__` and `__setattr__` for lazy attributes

Python's object hooks make it easy to writing generic code for gluing systems together. If a class defines `__getattr__`, that method is called every time an *attribute cannot be found in an object's instance dictionary.*

```
class LazyRecord:

    def __init__(self):
        self.exists = 5

    def __getattr__(self, name):
        value = f'Value for {name}'
```

```
        setattr(self, name, value)
        return value

data = LazyRecord()
print(data.__dict__) # prints {'exists': 5}
print(f'foo: {data.foo}') # prints 'Value for foo'
print(data.__dict__) # prints {'exists': 5, 'foo': 'Value for foo'}
```

The `exists` attribute is present in the instance dictionary, so `__getattr__` is never called for it. The `foo` attribute is not in the instance dictionary initially, so `__getattr__` is called the first time. But because the call to `__getattr__` also does a `setattr`, which populates `foo` in the instance dictionary, the second time `foo` is accessed, there is no call to `__getattr__`.

`__getattribute__` is called every time an attribute is accessed on an object, even in cases where *it does exist in the attribute dictionary.* This enables one to check global transaction state on every property access. *It is important, however, to note that such an operation can incur significant overhead and negatively impact performance.*

The `__setattr__` is always called every time an attribute is assigned on an instance, either directly or through the `setattr` built-in function.

Remember that one can avoid infinite recursion in `__getattribute__` and `__setattr__` by using methods from `super()` (i.e, the object class) to access instance attributes.

48: Validate subclasses with `__init_subclass__`

Often validation code runs in the `__init__` method when an object of the class's type is being instantiated. However, using metaclasses allows us to raise errors much earlier, for example when the module containing the class is first imported at the program startup.

A metaclass is defined as inheriting from `type`. In the default case, a metaclass receives the contents of associated `class` statements in its `__new__` method.

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        print(...)
        print(bases)
        print(class_dict)
        return type.__new__(meta, name, bases, class_dict)
```

Functionality can be added to the `Meta.__new__` method in order to validate all the parameters of an associated class before it is defined. See example pages 225 and 226, `ValidatePolygon`.

As an alternative to this, one can use the `__init_subclass__` special class method to provide the same level of validation as before:

```
class BetterPolygon:
    sides = None # must be specified by subclass

    def __init_subclass__(cls):
```

```
        super().__init_subclass__()
        if cls.sides < 3:
            raise ValueError(...)
```

This also has the advantage of easily providing multiple validations, see example in pages 227-229. `__init_subclass__` can be defined by multiple levels of a class hierarchy as long as the `super` built-in function is used to call any parent or sibling `__init_subclass__` definitions and it is also compatible with multiple inheritance.

49: Register class existence with `__init_subclass__`

`__init_subclass__` allows one to validate subclasses and register class existence:

```
class BetterRegisteredSerializable(BetterSerializable):
    def __init_subclass__(cls):
        super().__init_subclass__()
        register_class(cls)

class Vector1D(BetterRegisteredSerializable):
    def __init__(self, magnitude):
        super().__init__(magnitude)
        self.magnitude = magnitude
```

where `register_class` is defined in page 233.

50: Annotate class attributes with `__set_name__`

One feature enabled by metaclasses is the ability to modify or annotate properties after a class is defined but before the class is used.

Define a new class that represents a row in a customer database, with a corresponding property for each column of the table. First, define a descriptor class to define each of the properties:

```
class Field:

    def __init__(self, name):
        self.name = name
        self.internal_name = '_' + self.name

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)

class Customer:
    first_name = Field('first_name') # <--- redundancy
    last_name = Field('last_name')
```

```
    prefix = Field('prefix')
    suffix = Field('suffix')
```

However, the class definition of customer seems redundant because one is specifying the same information on the left-hand and right-hand sides.

The problem is that the order of operations in the `Customer` class definition is exactly the opposite of how it reads (L -> R). First, the `Field` constructor is called and then the return value is assigned to `Customer.first_name`. *There is no way for a* ***`Field`*** *instance to know upfront which class attribute it will be assigned to.*

This can be solved by using a metaclass, but one would need to be careful with the inheritance (see page 239). As an alternative, one can use `__set_name__`. This method is called on every descriptor instance when its containing class is defined. It receives as parameters the owning class that contains the descriptor instance and the attribute name to which the descriptor instance was assigned.

```
class Field:

    def __init__(self, name):
        self.name = name
        self.internal_name = '_' + self.name

    def __set_name__(self, owner, name):
        self.name = name
        self.internal_name = '_' + name

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)
```

Now, when the class `FixedCustomer` is constructed:

```
class FixedCustomer:
    first_name = Field()
    last_name = Field()

cust = FixedCustomer()
cust.first_name = 'Mersenne'
print(f"{cust.__dict__}") # returns {'first_name': 'Mersenne'}
```

51: Prefer class decorators over metaclasses for composable class extensions

Although metaclasses allow you to customize class creation in multiple ways, these still fall short of handling every situation that may arise. The example given there is decorating the methods of a class `TraceDict` to print arguments, return values and exceptions. The problems with this implementation are that:

1. each of the methods needs to be decorated
2. if the parent class of `TraceDict` adds a new method, then this new method needs to be implemented in `TraceDict` so that it is also decorated

One way to solve this problem is to use a metaclass to automatically decorate all methods of a class:

```
import types
from functools import wraps

trace_types = (...)

def trace_func(func):
    if hasattr(func, 'tracing'):
        return func

    @wraps(func)
    def wrapper(*args,**kwargs):
        result = None
        try:
            result = func(*args, **kwargs)
            return result
        except Exception as e:
            result = e
            raise
        finally:
            print(f'func.__name__'({args!r}, {kwargs!r}) -> ' , f'{result!r}')
    wrapper.tracing = True
    return wrapper

class TraceMeta(type):
    def __new__(meta, name, bases, class_dict):
        klass = super.__new__(meta, name, bases, class_dict)

        for key in dir(klass):
            value = getattr(klass, key)
            if isinstance(value, trace_types): # <-- see above
                wrapped = trace_func(value)
                setattr(klass, key, wrapped)
        return klass

class TraceDict(dict, metaclass = TraceMeta):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['hi'] # returns __getitem__({'hi': 1}, 'hi'), {}) -> 1
```

However, this approach is flawed if there are conflicting metaclasses (see page 244). To solve this problem, one can define *class decorators*.

```
from functools import wraps
```

```
trace_types = (...)

def trace_func(func):
    if hasattr(func, 'tracing'):
        return func

    @wraps(func)
    def wrapper(*args,**kwargs):
        result = None
        try:
            result = func(*args, **kwargs)
            return result
        except Exception as e:
            result = e
            raise
        finally:
            print(f'func.__name__'({args!r}, {kwargs!r}) -> ' , f'{result!r}')
    wrapper.tracing = True
    return wrapper

def trace(klass):
    for key in dir(klass):
        value = getattr(klass, key)
        if isinstance(value, trace_types):
            wrapped = trace_func(value)
            setattr(klass, key, wrapped)
    return klass

@trace
class TraceDict(dict):
    pass
```

When you are looking for composable ways to extend classes, class decorators are the best tool for the job. This is because metaclasses cannot be composed together easily, whereas many class decorators can be used to extend the same class without conflicts.

Chapter 7: Concurrency and parallelism

Concurrency enables a computer to do many different things seemingly at the same time, when it is actually interleaving the execution of the programs.

Parallelism involves actually doing many different things at the same time.

Threads support a relatively small amount of concurrency while coroutines enable vast numbers of concurrent functions.

52: Use `subprocesses` to manage child processes

Child processes started by Python are able to run in parallel, enabling you to use Python to consume all of the CPU cores of a machine and maximize the throughput of programs.

```
start = time.time()
sleep_procs = []

for _ in range(10):
    proc = subprocess.Popen(['sleep', '1'])
    sleep_procs.append(proc)

for proc in sleep_procs:
    proc.communicate()

end = time.time()

print(f'Finished in {end - start} seconds')
```

If these processes ran in sequence, the total delay would be 10s or more rather than the resulting ~1s.

53: Use threads for blocking I/O, avoid threads for parallelism

Global Interpreter Lock (GIL) is a mutual-exclusion lock (mutex) that prevents CPython from being affected by preemptive multithreading. However, it has a negative side-effect, in that Python only allows progress in one thread at a time, even though it supports multiple threads of execution.

Threads help handle blocking I/O by insulating a program from the time it takes for the operating system to respond to requests.

All the system calls will run in parallel from multiple Python threads even though they are limited by GIL. This works because Python threads release the GIL just before they make system calls and reacquire the GIL as soon as the system calls are done.

54: Use `lock` to prevent data races in thread

Although only one Python thread runs at a time, a thread's operation on data structures can be interrupted between any two byte-code instructions in the Python interpreter. As a result, the invariants of your data structures could be violated at practically any time because of these interruptions, leaving the program in a corrupted state.

See example in pages 236 and 237: the reasons why the result is incorrect are:

1. Python is actually performing multiple computations under the hood, in what looks like an atomic computation;
2. Due to the different threads and the multiple computations, the intermediate results can be overwritten;

To avoid this, you can use the context manager `from threading import Lock`.

55: Use `Queue` to coordinate work between threads

The `Queue` class has all the facilities you need to build robust pipelines: blocking operations (by default, see the example of page 242), buffer sizes (which prevents the queue from getting too big) and joining.

56: Know how to recognize when concurrency is necessary

57: Avoid creating new `Thread` instances for on-demand fan-out

Threads have many downsides:

- Costly to start and run when several of them are required;
- Require a significant amount of memory and special tools (like `Lock`) for coordination.

58: Understand how using `Queue` for concurrency requires refactoring

Instead of creating one thread per cell per generation (of the Game of Life), one can create a fixed number of worker threads upfront and have them do parallelized I/O as needed. *This will keep resource usage under control and eliminate the overhead of frequently starting new threads.*

`Queue` is better than using `Thread` but it is still a poor option because it limits the amount of I/O parallelism a program can leverage compared to alternative approaches provided by other built-in Python features.

59: Consider `ThreadPoolExecutor` when threads are necessary for concurrency

`ThreadPoolExecutor` is a good choice for situations where there is no asynchronous solution but there are better ways to *maximize I/O parallelism* in many cases.

60: Achieve highly concurrent I/O with `coroutines`

Key points:

- Functions that are defined using the `async` keyword are called coroutines. A caller can receive the result of a dependent coroutine by using the `await` keyword.
- Coroutines provide an efficient way to run tens of thousands of functions at *seemingly the same time*.
- Coroutines can use *fan-out* and *fan-in* in order to parallelize I/O, while also overcoming all of the problems associated with doing I/O in threads.

61: Know how to port threaded I/O to `asyncio`

Any code that previously interacted with the blocking `socket` instances has to be replaced with `asyncio` versions of similar functionality. All other lines in the function that require interaction with coroutines need to use `async` and `await` keywords as appropriate.

The `asyncio.create_task` function is used to enqueue the server for execution on the event loop so that it runs in parallel with the client when the `await` expression is reached.

62: Mix Threads and Coroutines to ease the transition to `asyncio`

One needs to be able to use threads for block I/O and coroutines for asynchronous I/O at the same time in a compatible way. In practice, one needs coroutines to start and wait on threads and threads to be able to run coroutines.

There are two approaches to incrementally convert the code to use `asyncio` and coroutines:

1. Top-down - start with the highest points (as the `main` entry points) and working down to the individual functions and classes (i.e, the leaves of the call hierarchy). *This approach can be useful when you maintain a lot of common modules that you use across many different programs.* By porting the entry points first, you can wait to port the common modules until you are using coroutines everywhere else:
    1. Change a top function to use `async def`
    2. Wrap all the calls that do I/O (*potentially blocking the event loop*) to use the `asyncio.run_in_executor`
    3. Ensure that the resources or callbacks used by `run_in_executor` invocations are properly synced (by using `Lock` or the `asyncio.run_coroutine_threadsafe`)
    4. Try to eliminate `get_event_loop` and `run_in_executor`
2. Bottom-up - transverses the call hierarchy in the opposite direction:
    1. Create a new `async` coroutine of each leaf function you are trying to port;
    2. Change the existing synchronous function so that these call the coroutine versions and *run the event loop instead of implementing any real behavior*;
    3. Move up the call hierarchy, make another layer of coroutines and replace existing calls to synchronous functions with the calls to the coroutines defined in step 1;
    4. Delete synchronous wrappers around coroutines created in step 2

What are the changes introduced in the `async` implementation?

1. A `asyncio` event `loop` is defined;
2. `write` is now an `async` function *of sorts*, which as a result does not need to use a lock, and instead uses `asyncio.run_coroutine_threadsafe`;
3. Rather than using `threads`, one uses `tasks`, which are handled in the `loop` through `run_in_executor` instead;
4. Finally, to combine the results, instead of using `thread.join()` one uses the `async` version `asyncio.gather(*tasks)`;

Step 4 is then the removal of the functions introduced in 1. (`get_event_loop`) and 3. (`run_in_executor`), by first pushing these one level lower (to `tail_async`) and then removing them from `run_task_mixed` -> so that the result is the async function `run_tasks`.

- The awaitable `run_in_executor` method of the `asyncio` event `loop` enables coroutines to run synchronous functions in `ThreadPoolExecutor` pools. This facilitates top-down migrations to `asyncio`.
- The `run_until_complete` method of the event loops enables synchronous code to run a coroutine until it finishes. The `run_coroutine_threadsafe`

provides the same functionality across thread boundaries.

63: Avoid blocking the `asyncio` event loop to maximize responsiveness

To write the most responsive program possible, one needs to minimize the potential system calls that are made from within the event loop.

64: Consider `concurrent.futures` for true parallelism

The `multiprocessing` built-in module, accessible via `concurrent.futures`, enables Python to utilize multiple CPU cores (through, for example, `ProcessPoolExecutor`) in parallel by running additional interpreters as child processes. Because the child processes are separate from the main interpreter, so their global interpreter locks are also separate.

What does `ProcessPoolExecutor` do?

1. Takes each input item from the arguments (e.g, `NUMBERS`) to `map`;
2. Serializes the item into binary data by using the `pickle` module;
3. Copies the serialized data from the main interpreter to a child interpreter over a local socket;
4. Deserializes the data back into Python object using `pickle`;
5. Imports the module containing the `gcd` function;
6. Runs the function on the input data in parallel with the other child processes;
7. Serializes the result back into binary data;
8. Copies the binary data through the socket;
9. Deserializes the binary data back into Python objects in the parent process;
10. Merges the results from multiple children into a single list to return;

Chapter 8: Robustness and Performance

65: Take advantage of each block in `try/except/else/finally`

`finally` blocks Use `try/finally` when you want exceptions to propagate up *but* you also want to run cleanup code *even when the exceptions* occur. Example: reliably closing file handles.

`else` blocks Use `try/except/else` to make it clear which exceptions will be handled by your code and which exceptions will propagate up. *When the try block does not raise an exception, the else block runs* (i.e, the `else` occurs on the `except`).

66: Consider `contextlib` and `with` statements for reusable `try/finally` behavior

How to use context managers correctly.

67: Use `datetime` instead of time for local clocks

68: Make `pickle` reliable with `copyreg`

The `copyreg` module lets you register the functions responsible for serializing and deserializing. `unpickle_game_state` calls the `GameState` constructor directly instead of using the `pickle` module default behavior of saving and restoring the attributes that belong to an object.

Any logic I need to adapt an old version of the class to a new version of the class can go in the `unpickle_game_state` function.

```
def unpickle_game_state(kwargs):
    version = kwargs.pop("version", 1)
    if version == 1:
        del kwargs["lives"]
    returns GameState(**kwargs)

def pickle_game_state(game_state):
    kwargs = game_state.__dict__
    kwargs["version"] = 2
    return unpickle_game_state , (kwargs,)

copyreg.pickle(GameState, pickle_game_state) # generates a serialized object
# with a stable identifier, so we could change the class name
# from GameState to BetterGameState without any issues
```

Note, however, that one cannot change the path of the module in which the `unpickle_game_state` function is present. Once a function is serialized, it must remain available on that import path for deserialization in the future.

69: Use `decimal` when precision is paramount

The `decimal` class provides fixed point math of 28 decimal places by default. One way to instantiate the `decimal` class is to use `str` as input, which ensures that there is no loss of precision.

70: Profile before optimizing

Operations you might assume would be slow are actually very fast (e.g, string manipulation, generators, …). Language features you might assume would be fast are actually very slow (e.g, attribute accesses, function calls).

71: Prefer `deque` for producer-consumer Queues

A FIFO queue is used when one function gathers values to process and another function handles them in the order in which they were received. Beware of using `list`'s `pop` for FIFO management because the performance of `pop` degrades quadratically.

72: Consider searching sorted sequences with `bisect`

73: Know how to use `heapq` for priority queues

A heap is a data structure that allows for a list of items to be maintained where the computational complexity of adding a new item or removing the smallest item has *logarithmic* computational complexity.

The heap-based priority queue scales much better - roughly `len(queue) * math.log(len(queue))` - without superlinearly degrading performance.

74: Consider `memoryview` and `bytearray` for zero-copy interactions with `bytes`

Chapter 9: Testing and debugging

80: Consider interactive debugging with `pdb`

- `where`: print the current execution call stack;
- `up`: move your scope up the execution call stack to the caller of the current function. This allows you to inspect the local variables in the higher levels of the program that led to the breakpoint;
- `down`: move your scope back down the execution call stack one level;

When you are done inspecting the current state, one can use five debugger commands to control the program's execution in different ways:

- `step`: run the program until the next line of execution in the program and return control back to the debugger prompt;
- `next`: run the program until the next line of execution in the current function and return … ;
- `return`: run the program until the current function returns and return …;
- `continue`: continue through the program until the next breakpoint is hit;
- `quit`: exit the debugger and end the program;

Post-mortem debugging can be used through `python -m pbc -c continue <program path>`.

81: Use `tracemalloc` to understand memory usage and leaks

Memory management in the default implementation of Python (CPython) uses reference counting, which ensures that as soon as all references to an object have expired, the referenced object is also cleared from memory, freeing up that space for other data. CPython also has a built-in cycle detector to ensure that self-referencing objects are eventually garbage-collected.

Chapter 10: Collaboration

90: Consider static analysis via `typing` to obviate bugs

- Type annotations slow you down when writing code. *A general strategy is to first write these without annotations, then write tests and then add type information where it is more valuable*;