CHAPTER

21 Language Questions

The limits of my language means the limits of my world.

— L. WITTGENSTEIN

21.1 Garbage Collection

What is garbage collection? Explain it in the context of Python.

Solution: Garbage collection is the process of finding data objects in a running program that cannot be accessed in the future, and to reclaim the resources, particularly memory, used by those objects. A garbage-collected language is one in which the garbage collection happens automatically—Java, C#, Python, and most scripting languages are examples. C is a notable example of a nongarbage-collected language—the programmer is responsible for knowing when to allocate and deallocate memory.

- Most garbage-collected languages used either reference counting (track the number of references to an object) or tracing (finding objects that are reachable by a sequence of references from certain "root" objects, and considering the rest as "garbage" and collecting them). Python uses reference counting, which has the benefit that it can immediately reclaim objects when the reference count goes to 0; the cost for this is the need for storing an additional integer-value per object. Tracing, which is used in Java, has the benefit that it can be performed in a separate thread, which makes it higher performance. A weakness of tracing is that when the garbage collector runs, it pauses all threads—this leads to nondeterministic performance (sporadic pauses).
- A challenge with reference counting is "reference cycles"—objects A and B reference each other, e.g., $A.u = B$ and $B.v = A$. The reference count never drops below 1, even if nothing else references A and B. The garbage collector periodically looks for these, and removes them.
- Garbage collectors used heuristics for speed. For example, empirically, recently created objects are more likely to be dead. Therefore, as objects are created, they are assigned to generations, and younger generations are examined first.

21.2 Closure

What does the following program print, and why?

```
increment_by_i = [lambda x: x + i for i in range(10)]

print(increment_by_i[3](4))
```

Solution: The program prints 13 (=9 + 4) rather than the 7 (=3 + 4) than might be expected. This is because the the functions created in the loop have the same scope. They use the same variable name, and consequently, all refer to the same variable, *i*, which is 10 at the end of the loop, hence the 13 (=9 + 4).

There are many ways to get the desired behavior. A reasonable approach is to return the lambda from a function, thereby avoiding the naming conflict.

```
def create_increment_function(x):
    return lambda y: y + x


increment_by_i = [create_increment_function(i) for i in range(10)]

print(increment_by_i[3](4))
```

21.3 Shallow and deep copy

Describe the differences between a shallow copy and a deep copy. When is each appropriate, and when is neither appropriate? How is deep copy implemented?

Solution: First, we note that assignment in Python does not copy—it simply a variable to a target. For objects that are mutable, or contain mutable items, a copy is needed if we want to change the copy without changing the original. For example, if `A = [1, 2, 4, 8]` and we assign `A` to `B`, i.e., `B = A`, then if we change `B`, e.g., `B.append(16)`, then `A` will also change.

There are two ways to make a copy: shallow and deep. These are implemented in the copy module, specifically as `copy.copy(x)` and `copy.deepcopy(x)`. A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original. A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

As a concrete example, suppose `A = [[1, 2, 3],[4, 5, 6]]` and `B = copy.copy(A)`. Then if we update `B` as `B[0][0] = 0`, `A[0][0]` also changes to `0`. However, if `B = copy.deepcopy(A)`, then the same update will leave `A` unchanged.

The difference between shallow and deep copying is only relevant for compound objects, that is objects that contain other objects, such as lists or class instances. If `A = [1, 2, 3]` and `B = copy.copy(A)`, then if we update `B` as `B[0] = 0`, `A` is unchanged.

Diving under the hood, deep copy is more challenging to implement since recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) result in infinite recursion in a naive implementation of deep copy. Furthermore, since a deep copy copies everything, it may copy too much, e.g., book-keeping data structures that should be shared even between copies. The `copy.deepcopy()` function avoids these problems by caching objects already copied, and letting user-defined classes override the copying operation. In particular, a class can override one or both of `__copy__()` and `__deepcopy__()`.

Copying is defensive—it may avoided if it's known that the client will not mutate the object. Similarly, if the object is itself immutable, e.g., a tuple, then there's no need to copy it.

21.4 Iterators and Generators

What is the difference between an iterator and a generator?

Solution: First, recall what an iterator is—it's any object that has `__iter__()` and `__next__()` methods. The first returns the iterator object itself, and is used is used in `for` and `in` statements. The second method returns the next value in the iteration—if there is no more items it raises the `StopIteration` exception. Here is an iterator that returns numbers that increment by a random amount between 0 and 1, stopping at a limit specified in the constructor.

```
# Iterable object, implements __iter__(), __next__().
class RandomIncrement():
    def __init__(self, limit):
        self._offset = 0.0
        self._limit = limit

    def __iter__(self):
        return self

    def __next__(self):
        self._offset += random.random()
        if (self._offset > self._limit):
            raise StopIteration()
        return self._offset

    # Call this to change the stop condition. It's safe to interleave this with
    # usage of the iterator.
    def increment_limit(self, increment_amount):
        self._limit += increment_amount
```

A generator is an easy way to create iterators. As a concrete example, here is the same iterator expressed as a generator. Note the use of the keyword `yield`.

```
def random_iterator(limit):
    offset = 0
    while True:
        offset += random.random()
        if (offset > limit):
            raise StopIteration()
        yield offset
```

A generator uses the function call stack to implicitly store the state of the iterator—this can simplify the writing of an iterator compared to writing the same iterator as an explicit class; it also helps readability.

Every generator is an iterator, but the converse is not true. In particular, an iterator can be a full-blown class, and can therefore offer additional functionality. For example, it's easy to add a method to the iterator class above to change the iteration limit—this is impossible with the generator.

21.5 @decorator

Explain what a decorator is, with an example that shows why its useful.

Solution: First, recall that functions are first class objects in Python, that is, they can be passed as arguments to other functions, and returned by functions. Additionally, functions can be defined within other functions. In certain contexts, these facts can greatly simplify writing code. For example, here is code that prints the time take to run a function.

```
def time_function(f):
    '''
    Print how long a function takes to compute.
    '''
    begin = time.time()
    result = f()
    end = time.time()
    print("Function call took " + str(end - begin) + " seconds to execute.")
    return result


def foo():
    print("I am foo()")


def ackermann(m, n):
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        return ackermann(m - 1, ackermann(m, n - 1))


time_function(foo)
# functools.partial() take a function and positional arguments to it and
# creates a new function with those arguments preassigned to the specified
# values.
time_function(functools.partial(ackermann, 3, 4))
```

Python provides syntactic sugaring to simplify this process in the form of the `@decorator` construct. The annotation `@time_function` amounts to the following: `foo = @time_function(foo)`.

```
def time_function(f):
    def wrapper(*args, **kwargs):
        begin = time.time()
        result = f(*args, **kwargs)
        end = time.time()
        print("Function call with arguments {all_args} took ".format(
            all_args="\t".join((str(args), str(kwargs)))) + str(end - begin) +
              " seconds to execute.")
        return result

    return wrapper


@time_function
def foo():
    print("I am foo()")
```

```
@time_function
def ackermann(m, n):
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        return ackermann(m - 1, ackermann(m, n - 1))


@time_function
def bar(*args, **kwargs):
    print(sum(args) * sum(kwargs.values()))
```

Other common uses of decorators are checking access control on function backing up REST endpoints, and adding a delay to the end of a function to rate-limit it.

21.6 List vs tuple

In what ways are lists and tuples similar, and in what ways are they different?

Solution: Lists, e.g., `[6, 28, 496, 8128]` and tuples, e.g., `(3.14, "pi", [True, False, True])` are similar in that both represent sequences. Both use the same syntax to access, the i-th element of the sequence, and both support the `in` operator for membership checking.

The key difference between a tuple and a list is that a tuple is immutable—you cannot change the element at index i, or add/delete from a tuple, all of which are possible with a list.

There are numerous benefits of immutability, the most important of which immutable objects are more container-friendly and thread-safe. Elaborating on container-friendliness, if a mutable object is inserted in a set, changed, and then the changed objected is looked up, the lookup will fail (because the hashcode has changed from when it was added to the set). For this reason, tuples are can be put in sets, and used as map keys, but lists cannot.

For the sake of completeness, it's worth pointing out that it's possible to have a mutable object in a tuple, and as a result such a tuple is not longer immutable, as illustrated below:

```
class A():
    def __init__(self, x):
        self.x = x

    def __eq__(self, other):
        return isinstance(other, A) and self.x == other.x

    def __hash__(self):
        return self.x * 113 + 119


u = A(42)
v = A(42)
U = (u, )
V = (v, )
```

```
S = set([U])
print(U in S)  # Prints True.
print(V in S)  # Also prints True, since we implemented equals and hash.
u.x = 28
print(U in S)  # Prints False, since u has changed.
```

Some less important differences include the following:

- Conventionally, tuples are used to represent heterogeneous groups of values, where lists are used for homogeneous values, as in the example above.
- Tuples are slightly faster to build and access, and have a smaller memory footprint.

21.7 *ARGS AND *KWARGS

What are `*args` and `*kwargs`? Where are they appropriate?

Solution: Both are used to pass a variable number of arguments to a function. The first, `*args`, is used to pass a variable length argument list, e.g.,

```
def foo(u, v, *args):
    print('u,v = ' + str((u, v)))
    for i in range(len(args)):
        print('args {0} = {1}'.format(str(i), str(args[i])))


foo(1, 'euler', 2.71, [6, 28])
'''
Alternative call.
With the *, foo gets 4 arguments, not 3. Prints
u,v = (1, 'euler')
args 0 = 2.71
args 1 = [6, 28]
'''
args = (2.71, [6, 28])
foo(1, 'euler', *args)
'''
Without the *, foo gets 3 arguments, not 4. Prints
u,v = (1, 'euler')
args 0 = (2.71, [6, 28])
'''
foo(1, 'euler', args)
```

Some points:

- It's not important that the parameter referenced in the function be called `args`—it could just as well be called `A` or `varargs` (though `args` is idiomatic);
- it is important that the * be specified;
- the * argument must appear after all the regular arguments (if any); and
- * can be used at the call site too.

The second, `**kwargs`, is used when passing a variable number of keyword arguments to a function.

```
def foo(u, v, *args, **kwargs):
    print('u,v = ' + str((u, v)))
    for i in range(len(args)):
        print('args {0} = {1}'.format(str(i), str(args[i])))
    for (keyword, value) in kwargs.items():
        print('keyword,value =  ' + str((keyword, value)))


foo(1, 'euler', 2.71, [6, 28], name='cfg', rank=1)
# Alternate call
args = (2.71, [6, 28])
kwargs = {'name': 'cfg', 'rank': 1}
foo(1, 'euler', *args, **kwargs)
```

Some points:

- It's not important that the parameter referenced in the function be called `args`—it could just as well be called `D` or `argdict`;
- it is important is the `**` be specified;
- the `**` argument must appear after all the regular named parameters and the `*` argument (if any);
- `**` can be used at the call site too; and
- keyword arguments are quite different from named arguments, wherein the names of the arguments are specified in the function itself.

21.8 Python code

Rewrite the following program using a Pythonic style.

```
def compute_top_k_variance(students, scores, k):
    '''
    students and scores are equal length lists of strings and floats,
    respectively. The function computes for each string that appears at least
    k times in the list the variance of the top k scores that correspond to it.
    Strings that appear fewer than k times are not considered.
    '''
    counts = {}
    for i in range(len(students)):
        if students[i] not in counts:
            counts[students[i]] = 1
        else:
            counts[students[i]] += 1

    all_scores = {}
    for key in counts:
        if counts[key] >= k:
            all_scores[key] = []

    for i in range(len(students)):
        if students[i] in all_scores:
            all_scores[students[i]].append(scores[i])
```

```
    top_k_scores = {}
    for key in all_scores:
        sorted_scores = sorted(all_scores[key])
        top_k_scores[key] = []
        for i in range(k):
            top_k_scores[key].append(sorted_scores[len(sorted_scores) - 1 - i])

    result = {}
    for key in top_k_scores:
        total = 0
        for score in top_k_scores[key]:
            total += score
        mean = total / k
        variance = 0
        for score in top_k_scores[key]:
            variance = variance + (score - mean) * (score - mean)
        result[key] = variance

    return result
```

Solution: The functions `zip()`, `functools.reduce()`, `heapq.nlargest()`, and `sum()`, directly replace boilerplate loops. The `collections.defaultdict` container and dictionary comprehension remove the need for explicit initialization.

```
def compute_top_k_variance(students, scores, k):
    all_scores = collections.defaultdict(list)
    for student, score in zip(students, scores):
        all_scores[student].append(score)

    top_k_scores = {
        student: heapq.nlargest(k, scores)
        for student, scores in all_scores.items() if len(scores) >= k
    }

    return {
        student: functools.reduce(
            lambda variance, score: variance + (score - mean)**2, scores, 0)
        for student, scores, mean in (
            (student, scores, sum(scores) / k)
            for student, scores in top_k_scores.items())
    }
```

21.9 Exception Handling

Briefly describe exception handling in Python, paying special attention to the roles played by `try`, `except`, `else`, `finally`, and `raise`. Rewrite the following program using exceptions to make it more robust.

```
def get_col_sum(filename, col):
    # May raise IOError.
    csv_file = open(filename)
    csv_reader = csv.reader(csv_file)
```

```
    running_sum = 0
    for row in csv_reader:
        value = row[col]
        running_sum += int(value)
    csv_file.close()
    print("Sum = " + str(running_sum))
```

Solution: A statement or expression it may result in an error when an attempt is made to execute it. Such errors are called exceptions. They do not always entail exiting the program. For example, if a user enters a filename, and no corresponding file is found, or the user does not have read permissions, the user can be prompted again. Such checks can be made using a `try` block.

The `try` block can be followed by an `except` block, a `finally` block, or an `except` block followed by a `finally` block or an `except` block followed by an `else` block followed by a `finally` block.

If a `finally` block exists, it is always executed, regardless of whether an exception was raised and/or caught. Here is an example. Note that there is no `except` block—any exception raised in the `try` block is propagated up.

```
def get_value(filename, key):
    handle = open(filename)
    try:
        file_contents = handle.read()
        js_text = json.loads(file_contents)
        return js_text[key]
    finally:
        # Prevent resource leak if there is an error parsing file_contents.
        handle.close()
```

The code below shows basic `try/except` usage. Note the duplicated code used to avoid the resource leak.

```
def get_value(filename, key):
    handle = open(filename)
    try:
        file_contents = handle.read()
        js_text = json.loads(file_contents)
        handle.close()
        return (True, js_text[key])
    except ValueError:
        handle.close()
        return (False, )
```

The duplicated code can be avoided by using `try/except/finally`.

```
def get_value(filename, key):
    handle = open(filename)
    try:
        file_contents = handle.read()
        js_text = json.loads(file_contents)
        return (True, js_text[key])
    except ValueError:
        return (False, )
    finally:
        handle.close()
```

Here's the original program rewritten using `try/except/else/finally` blocks to make it more robust. In particular, the `else` shows what code is executed in the absence of exceptions. Note how we use `raise` to create/propagate exceptions upwards.

```
class ColSumCsvParseException(Exception):
    def __init__(self, *args):
        Exception.__init__(self, *args)
        self.line_number = args[1]


def get_col_sum(filename, col):
    # may raise IOError, will propagate to caller
    csv_file = open(filename)
    csv_reader = csv.reader(csv_file)
    running_sum = line_number = 0
    try:
        for row in csv_reader:
            if col >= len(row):
                raise IndexError("Not enough entries in row " + str(row))
            value = row[col]
            # We will skip rows for which the corresponding columns cannot be
            # parsed to an int, logging the fact.
            try:
                running_sum += int(value)
            except ValueError:
                print("Cannot convert " + value + " to int, ignoring")
            line_number += 1
    except csv.Error:
        # Programs should raise exceptions appropriate to their level of
        # abstraction, so we propagate the csv.Error upwards as a
        # ColSumCsvParseException.
        print("In csv.Error handler")
        raise ColSumCsvParseException("Error processing csv", line_number)
    else:
        print("Sum = " + str(running_sum))
    finally:
        # Ensure there is no resource leak.
        csv_file.close()
        return running_sum
```

There are a number of built-in exception types in Python, e.g., `IndexError`, `FloatingPointError`, `EnvironmentError`, `ValueError`, etc., and these should be used whenever possible.

21.10 Scoping

Explain the rules for variable scope.

Solution: There are two (nonexclusive) possibilities: the variable appears in an expression, and the variable is being assigned to.

When the variable appears in an expression, Python searches for it in the following order:

(1.) The current function.

(2.) Enclosing scopes, e.g., containing functions.

(3.) The module containing the code (also referred to as the global scope)

(4.) The built-in scope, e.g., `open`

(A `NameError` is raised if none of these contain a defined variable with the given name.)

In the simplest case, the variable is defined already in the current scope, in which case the assignment happens to it. Otherwise, if the assignment happens outside a function, it defines a new global variable; if it happens within a function, it defines a new variable whose scope is the function that contains the assignment.

To make an assignment to a variable not defined in the current scope within a function happen to a global variable, define a new variable in the current scope, that variable should be declared as `global` in the function. In Python3, to have an assignment to a variable in nested function use an enclosing function's scope, use `nonlocal`: this will make the program search outwards through the enclosing scopes (but not to the module level) for that variable.

The scoping rules seek to prevent variables local to a function from polluting the containing functions and module. They can lead to scoping bugs though, so as a rule names should be kept distinct.

Here is a program that illustrates these rules.

```
x, y, z = 'global-x', 'global-y', 'global-z'


def basic_scoping():
    print(x)  # global-x
    y = 'local-y'
    global z
    z = 'local-z'


basic_scoping()
print(x, y, z)  # global-x global-y local-z


def inner_outer_scoping():
    def inner1():
        print(x)  # outer-x

    def inner2():
        x = 'inner2-x'
        print(x)  # inner2-x

    def inner3():
        nonlocal x
        x = 'inner3-x'
        print(x)  # inner3-x

    x = "outer-x"
    inner1(), inner2(), inner3()
    print(x)  # inner3-x


inner_outer_scoping()
print(x, y, z)  # global-x global-y local-z
```

```
def outer_scope_error():
    def inner():
        try:
            x = x + 321
        except NameError:
            print('Error: x is local, and so x + 1 is not defined yet')

    x = 123
    inner()


outer_scope_error()  # prints 'Error: ...'


def outer_scope_array_no_error():
    def inner():
        x[0] = -x[0]  # x[0] isn't a variable, it's resolved from outer x.

    x = [314]
    inner()
    print(x[0])  # -314


outer_scope_array_no_error()
```

21.11 Function arguments

What are positional, keyword, and default arguments to a function. What does the following program output, and why?

```
def foo(x=[]):
    x.append(1)
    return x


res = foo()
print(res)
res = foo()
print(res)
```

Solution: On the calling side, some function arguments can be specified by name. These arguments have come after all of the unnamed arguments (also known as positional arguments). Consider the following function.

```
def foo(x, y, z):
    return x * x + y * y + z * z
```

It can be called using positional arguments, or keyword arguments as follows.

```
foo(1, 2, 3)
```

```
foo(x=1, y=2, z=3)
foo(1, y=2, z=3)
foo(1, z=3, y=3)
foo(z=3, y=3, x=1)
# This is a syntax error, keyword arguments must follow positional arguments.
foo(x=1, 2, z=3)
# This is a syntax error, a variable cannot be assigned twice.
foo(1, x=2, z=3, y=4)
```

Keywords arguments have the following benefits.

- The function call is much clearer—it's harder to mistakenly exchange two values.
- They make it easier to refactor a function into having more arguments. Specifically, the combination of keyword arguments and default values means that old code does not have to be touched even when new arguments are added.

When a function is defined, its arguments can be specified to have default values; if the function is called without the argument, the argument gets its default value. There must be default values for all arguments which are not specified in calls to the function.

One subtlety with default arguments is that they are evaluated once, specifically when the module is loaded, and are shared across all callers. For this reason, the program given in the example prints `[1,1]` after the second call—the first call updates the default argument to `[1]`. As a rule, mutable arguments should have `None` as their default values. Then the function can test the argument in its body—if it is `None`, the function can assign it to the desired default value. See the program below for an example.

```
def read_file_as_array_buggy(filename, default=[]):
    try:
        filehandle = open(filename)
        return filename.read().split()
    except Exception:
        return default


def read_file_as_array_works(filename, default=None):
    if default is None:
        default = []
    try:
        filehandle = open(filename)
        return filename.read().split()
    except Exception:
        return default


A = read_file_as_array_buggy("does_not_exist.txt")
A.append("first")
B = read_file_as_array_buggy("does_not_exist.txt")
B.append("second")
print(B)  # ['first', 'second']
A = read_file_as_array_works("does_not_exist.txt")
A.append("first")
B = read_file_as_array_works("does_not_exist.txt")
B.append("second")
```

```
print(B)  # ['second']
```

CHAPTER

22

Object-Oriented Design

One thing expert designers know not to do is solve every problem from first principles.

— *"Design Patterns: Elements of Reusable Object-Oriented Software,"* E. GAMMA, R. HELM, R. E. JOHNSON, AND J. M. VLISSIDES, 1994

A class is an encapsulation of data and methods that operate on that data. Classes match the way we think about computation. They provide encapsulation, which reduces the conceptual burden of writing code, and enable code reuse, through the use of inheritance and polymorphism. However, naive use of object-oriented constructs can result in code that is hard to maintain.

A design pattern is a general repeatable solution to a commonly occurring problem. It is not a complete design that can be coded up directly—rather, it is a description of how to solve a problem that arises in many different situations. In the context of object-oriented programming, design patterns address both reuse and maintainability. In essence, design patterns make some parts of a system vary independently from the other parts.

Adnan's Design Pattern course material, available freely online, contains lecture notes, homeworks, and labs that may serve as a good resource on the material in this chapter.

22.1 TEMPLATE METHOD VS. STRATEGY

Explain the difference between the template method pattern and the strategy pattern with a concrete example.

Solution: Both the template method and strategy patterns are similar in that both are behavioral patterns, both are used to make algorithms reusable, and both are general and very widely used. However, they differ in the following key way:

- In the template method, a skeleton algorithm is provided in a superclass. Subclasses can override methods to specialize the algorithm.
- The strategy pattern is typically applied when a family of algorithms implements a common interface. These algorithms can then be selected by clients.

As a concrete example, consider a sorting algorithm like quicksort. Two of the key steps in quicksort are pivot selection and partitioning. Quicksort is a good example of a template method—subclasses can implement their own pivot selection algorithm, e.g., using randomized median finding or selecting an element at random, and their own partitioning method, e.g., using the DNF partitioning algorithms in Solution 5.1 on Page 40.

Since there may be multiple ways in which to sort elements, e.g., student objects may be compared by GPA, major, name, and combinations thereof, it's natural to make the comparison operation used by the sorting algorithm an argument to quicksort. One way to do this is to pass

quicksort an object that implements a compare method. These objects constitute an example of the strategy pattern, as do the objects implementing pivot selection and partitioning.

There are some other smaller differences between the two patterns. For example, in the template method pattern, the superclass algorithm may have "hooks"—calls to placeholder methods that can be overridden by subclasses to provide additional functionality. Sometimes a hook is not implemented, thereby forcing the subclasses to implement that functionality; some times it offers a "no-operation" or some baseline functionality. There is no analog to a hook in a strategy pattern.

Note that there's no relationship between the template method pattern and template meta-programming (a form of generic programming favored in C++).

22.2 Observer pattern

Explain the observer pattern with an example.

Solution: The observer pattern defines a one-to-many dependency between objects so that when one object changes state all its dependents are notified and updated automatically.

The observed object must implement the following methods.

- Register an observer.
- Remove an observer.
- Notify all currently registered observers.

The observer object must implement the following method.

- Update the observer. (Update is sometimes referred to as notify.)

As a concrete example, consider a service that logs user requests, and keeps track of the 10 most visited pages. There may be multiple client applications that use this information, e.g., a leaderboard display, ad placement algorithms, recommendation system, etc. Instead of having the clients poll the service, the service, which is the observed object, provides clients with register and remove capabilities. As soon as its state changes, the service enumerates through registered observers, calling each observer's update method.

Though most readily understood in the context of a single program, where the observed and observer are objects, the observer pattern is also applicable to distributed computing.

22.3 Push vs. pull observer pattern

In the observer pattern, subjects push information to their observers. There is another way to update data—the observers may "pull" the information they need from the subject. Compare and contrast these two approaches.

Solution: Both push and pull observer designs are valid and have tradeoffs depending on the needs of the project. With the push design, the subject notifies the observer that its data is ready and includes the relevant information that the observer is subscribing to, whereas with the pull design, it is the observer's job to retrieve that information from the subject.

The pull design places a heavier load on the observers, but it also allows the observer to query the subject only as often as is needed. One important consideration is that by the time the observer retrieves the information from the subject, the data could have changed. This could be a positive or negative result depending on the application. The pull design places less responsibility on the subject for tracking exactly which information the observer needs, as long as the subject knows

when to notify the observer. This design also requires that the subject make its data publicly accessible by the observers. This design would likely work better when the observers are running with varied frequency and it suits them best to get the data they need on demand.

The push design leaves all of the information transfer in the subject's control. The subject calls update for each observer and passes the relevant information along with this call. This design seems more object-oriented, because the subject is pushing its own data out, rather than making its data accessible for the observers to pull. It is also somewhat simpler and safer in that the subject always knows when the data is being pushed out to observers, so you don't have to worry about an observer pulling data in the middle of an update to the data, which would require synchronization.

22.4 Singletons and Flyweights

Explain the differences between the singleton pattern and the flyweight pattern. Use concrete examples.

Solution: The singleton pattern ensures a class has only one instance, and provides a global point of access to it. The flyweight pattern minimizes memory use by sharing as much data as possible with other similar objects. It is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory.

A common example of a singleton is a logger. There may be many clients who want to listen to the logged data (console, file, messaging service, etc.), so all code should log to a single place.

A common example of a flyweight is string interning—a method of storing only one copy of each distinct string value. Interning strings makes some string processing tasks more time- or space-efficient at the cost of requiring more time when the string is created or interned. The distinct values are usually stored in a hash table. Since multiple clients may refer to the same flyweight object, for safety flyweights should be immutable.

There is a superficial similarity between singleton and flyweight: both keep a single copy of an object. There are several key differences between the two:

- Flyweights are used to save memory. Singletons are used to ensure all clients see the same object.
- A singleton is used where there is a single shared object, e.g., a database connection, server configurations, a logger, etc. A flyweight is used where there is a family of shared objects, e.g., objects describing character fonts, or nodes shared across multiple binary search trees.
- Flyweight objects are invariable immutable. Singleton objects are usually not immutable, e.g., requests can be added to the database connection object.
- The singleton pattern is a creational pattern, whereas the flyweight is a structural pattern.

In summary, a singleton is like a global variable, whereas a flyweight is like a pointer to a canonical representation.

Sometimes, but not always, a singleton object is used to create flyweights—clients ask the singleton for an object with specified fields, and the singleton checks its internal pool of flyweights to see if one exists. If such an object already exists, it returns that, otherwise it creates a new flyweight, add it to its pool, and then returns it. (In essence the singleton serves as a gateway to a static factory.)

What is the difference between a class adapter and an object adapter?

Solution: The adapter pattern allows the interface of an existing class to be used from another interface. It is often used to make existing classes work with others without modifying their source code.

There are two ways to build an adapter: via subclassing (the class adapter pattern) and composition (the object adapter pattern). In the class adapter pattern, the adapter inherits both the interface that is expected and the interface that is pre-existing. In the object adapter pattern, the adapter contains an instance of the class it wraps and the adapter makes calls to the instance of the wrapped object.

Here are some remarks on the class adapter pattern.

- The class adapter pattern allows re-use of implementation code in both the target and adaptee. This is an advantage in that the adapter doesn't have to contain boilerplate pass-throughs or cut-and-paste reimplementation of code in either the target or the adaptee.
- The class adapter pattern has the disadvantages of inheritance (changes in base class may cause unforeseen misbehaviors in derived classes, etc.). The disadvantages of inheritance are made worse by the use of two base classes, which also precludes its use in languages like Java (prior to Java 1.8) that do not support multiple inheritance.
- The class adapter can be used in place of either the target or the adaptee. This can be advantage if there is a need for a two-way adapter. The ability to substitute the adapter for adaptee can be a disadvantage otherwise as it dilutes the purpose of the adapter and may lead to incorrect behavior if the adapter is used in an unexpected manner.
- The class adapter allows details of the behavior of the adaptee to be changed by overriding the adaptee's methods. Class adapters, as members of the class hierarchy, are tied to specific adaptee and target concrete classes.

As a concrete example of an object adapter, suppose we have legacy code that returns objects of type stack. Newer code expects inputs of type deque, which is more general than stack (but does not subclass stack). We could create a new type, stack-adapter, which implements the deque methods, and can be used anywhere deque is required. The stack-adapter class has a field of type stack—this is referred to as object composition. It implements the deque methods with code that uses methods on the composed stack object. Deque methods that are not supported by the underlying stack throw unsupported operation exceptions. In this scenario, the stack-adapter is an example of an object adapter.

Here are some comments on the object adapter pattern.

- The object adapter pattern is "purer" in its approach to the purpose of making the adaptee behave like the target. By implementing the interface of the target only, the object adapter is only useful as a target.
- Use of an interface for the target allows the adaptee to be used in place of any prospective target that is referenced by clients using that interface.
- Use of composition for the adaptee similarly allows flexibility in the choice of the concrete classes. If adaptee is a concrete class, any subclass of adaptee will work equally well within the object adapter pattern. If adaptee is an interface, any concrete class implementing that interface will work.

- A disadvantage is that if target is not based on an interface, target and all its clients may need to change to allow the object adapter to be substituted.

Variant: The UML diagrams for decorator, adapter, and proxy look identical, so why is each considered a separate pattern?

22.6 Creational Patterns

Explain what each of these creational patterns is: builder, static factory, factory method, and abstract factory.

Solution: The idea behind the builder pattern is to build a complex object in phases. It avoids mutability and inconsistent state by using an mutable inner class that has a build method that returns the desired object. Its key benefits are that it breaks down the construction process, and can give names to steps. Compared to a constructor, it deals far better with optional parameters and when the parameter list is very long.

A static factory is a function for construction of objects. Its key benefits are as follow: the function's name can make what it's doing much clearer compared to a call to a constructor. The function is not obliged to create a new object—in particular, it can return a flyweight. It can also return a subtype that's more optimized, e.g., it can choose to construct an object that uses an integer in place of a Boolean array if the array size is not more than the integer word size.

A factory method defines interface for creating an object, but lets subclasses decide which class to instantiate. The classic example is a maze game with two modes—one with regular rooms, and one with magic rooms. The program below uses a template method, as described in Problem 22.1 on Page 333, to combine the logic common to the two versions of the game.

```
from abc import ABC, abstractmethod


class Room(ABC):
    @abstractmethod
    def connect(self, room2):
        pass


class MazeGame(ABC):
    @abstractmethod
    def make_room(self):
        print("abstract make_room")
        pass

    def addRoom(self, room):
        print("adding room")

    def __init__(self):
        room1 = self.make_room()
        room2 = self.make_room()
        room1.connect(room2)
        self.addRoom(room1)
        self.addRoom(room2)
```

This snippet implements the regular rooms. This snippet implements the magic rooms.

```
class MagicMazeGame(MazeGame):
    def make_room(self):
        return MagicRoom()


class MagicRoom(Room):
    def connect(self, room2):
        print("Connecting magic room")
```

Here's how you use the factory to create regular and magic games.

```
ordinary_maze_game = ordinary_maze_game.OrdinaryMazeGame()
magic_maze_game = magic_maze_game.MagicMazeGame()
```

A drawback of the factory method pattern is that it makes subclassing challenging.

An abstract factory provides an interface for creating families of related objects without specifying their concrete classes. For example, a class `DocumentCreator` could provide interfaces to create a number of products, such as `createLetter()` and `createResume()`. Concrete implementations of this class could choose to implement these products in different ways, e.g., with modern or classic fonts, right-flush or right-ragged layout, etc. Client code gets a `DocumentCreator` object and calls its factory methods. Use of this pattern makes it possible to interchange concrete implementations without changing the code that uses them, even at runtime. The price for this flexibility is more planning and upfront coding, as well as and code that may be harder to understand, because of the added indirections.

22.7 Libraries and design patterns

Why is there no library of design patterns so a developers do not have to write code every time they want to use them?

Solution: There are several reasons that design patterns cannot be delivered as a set of libraries. The key idea is that patterns cannot be cleanly abstracted from the objects and the processes they are applicable to. More specifically, libraries provide the implementations of algorithms. In contrast, design patterns provide a higher level understanding of how to structure classes and objects to solve specific types of problems. Another difference is that it's often necessary to use combinations of different patterns to solve a problem, e.g., Model-View-Controller (MVC), which is commonly used in UI design, incorporates the Observer, Strategy, and Composite patterns. It's not reasonable to come up with libraries for every possible case.

Of course, many libraries take advantage of design patterns in their implementations: sorting and searching algorithms use the template method pattern, custom comparison functions illustrate the strategy pattern, string interning is an example of the flyweight pattern, typed-I/O shows off the decorator pattern, etc.

Variant: Give examples of commonly used library code that use the following patterns: template, strategy, observer, singleton, flyweight, static factory, decorator, abstract factory.

Variant: Compare and contrast the iterator and composite patterns.