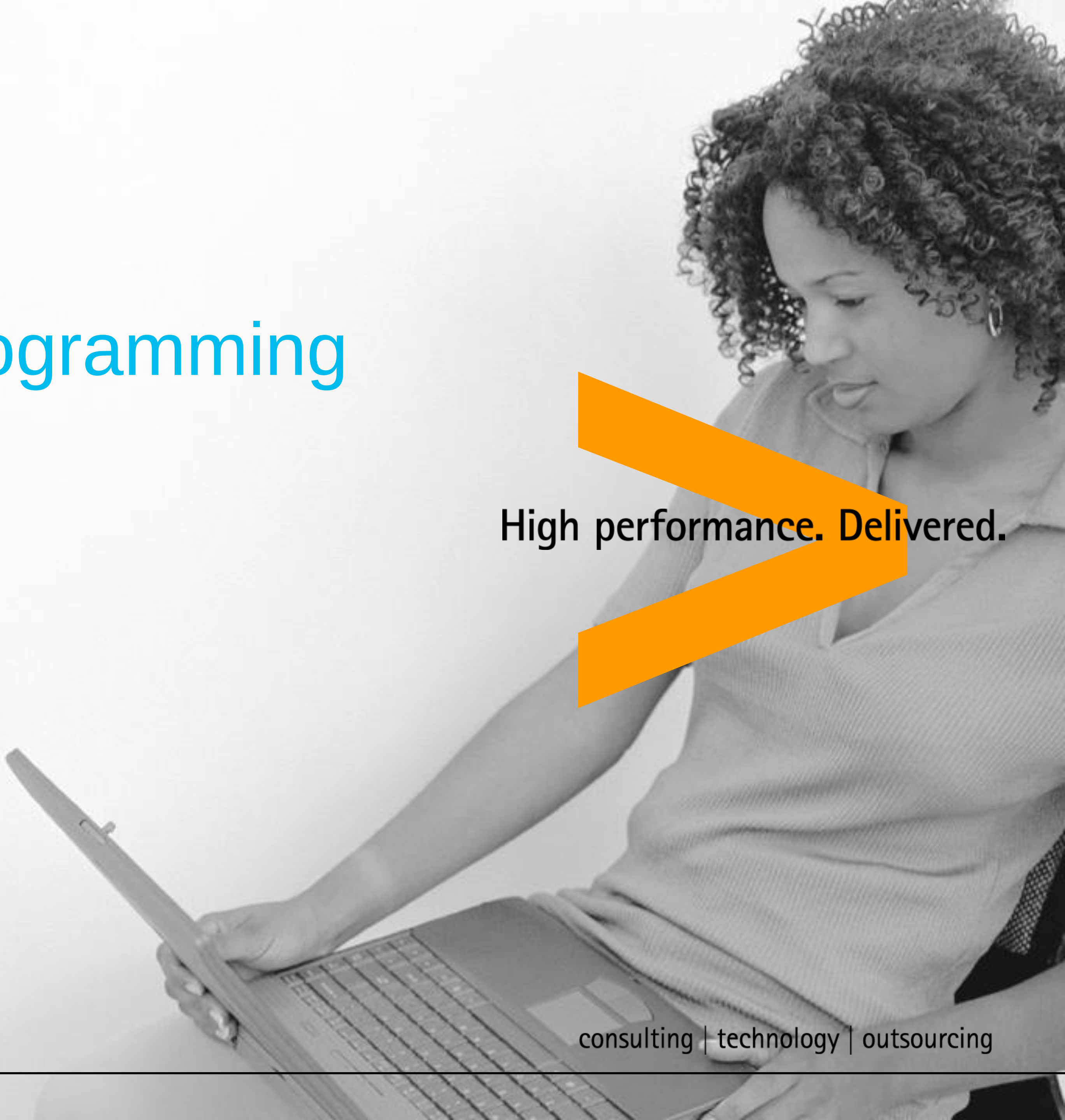
Python Programming



High performance. Delivered.

consulting | technology | outsourcing

Python

Goals

- Regular Expressions
- Decorators
- Iterators and Generators
- Collections Framework
- Meta Class
- Sockets

Python

Goals

- Networking
- Threading
- Processes
- Signal Handling
- Unit Testing
- Image Handling

Python

Goals

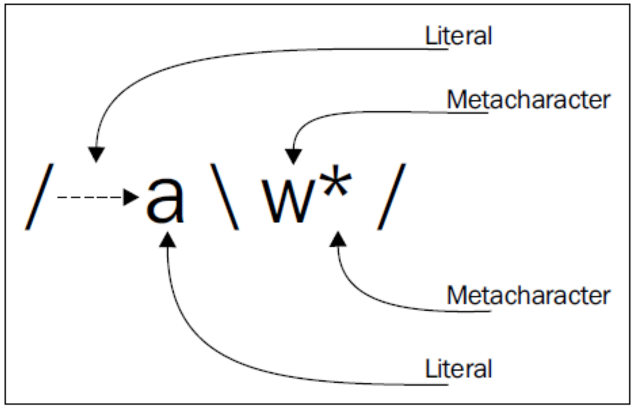
- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Regular Expressions

- A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern.
- Regular expressions are widely used in UNIX world.
- The module re provides full support for Perl-like regular expressions in Python.
- The re module raises the exception re.error if an error occurs while compiling or using a regular expression

Regular Expressions

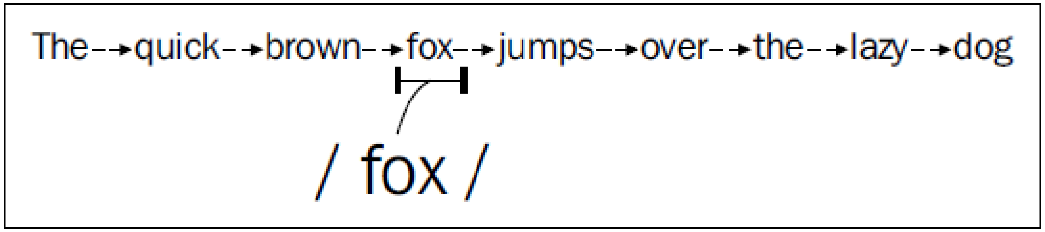
- A regular expression is a pattern of text that consists of ordinary characters (for example, letters a through z or numbers 0 through 9) and special characters known as meta characters.
- This pattern describes the strings that would match when applied to a text.



Regex using literals and metacharacters

Regular Expressions - Literals

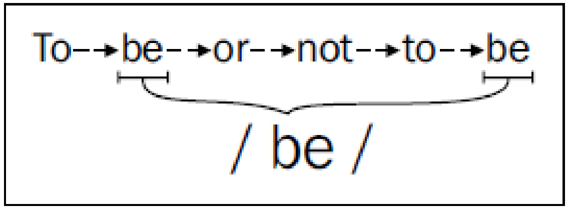
- Literals are the simplest form of pattern matching in regular expressions.
- They will simply succeed whenever that literal is found.
- If we apply the regular expression /fox/ to search the phrase The quick brown fox jumps over the lazy dog, we will find one match:



Searching using a regular expression

Regular Expressions - Literals

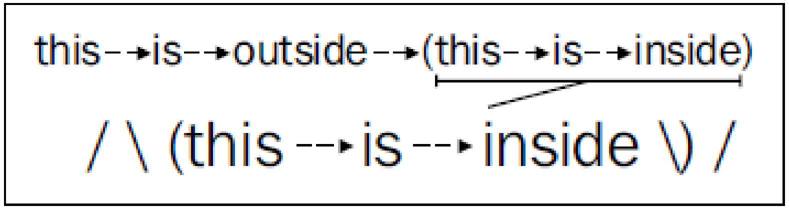
- However, we can also obtain several results instead of just one, if we apply the
- regular expression /be/ to the following phrase To be, or not to be:



Multiple results searching with regex

Regular Expressions - Literals

- Using the backslash method, we can convert the previous expression to /\(this is inside\)/ and apply it again to the same text to have the parentheses included in the result:



this→is→outside→(this→is→inside)

/ \ (this→is→inside \) /

Escaped metacharacters in regex

Regular Expression

- In regular expressions, there are twelve meta characters that should be escaped if they are to be used with their literal meaning:
- Backslash \
- Caret ^
- Dollar sign $
- Dot .

Regular Expression

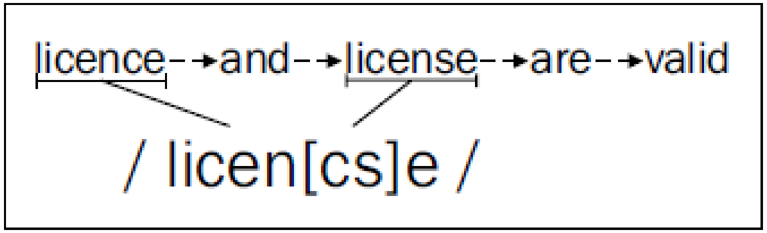
Pipe symbol |

- Question mark ?
- Asterisk *
- Plus sign +
- Opening parenthesis (
- Closing parenthesis)
- Opening square bracket [
- The opening curly brace {

Regular Expressions Character classes

- The character classes (also known as character sets) allow us to define a character that will match if any of the defined characters on the set is present.
- To define a character class, we should use the opening square bracket meta character [, then any accepted characters, and finally close with a closing square bracket].
- For instance, let's define a regular expression that can match the word "license" in British and American English written form:

Regular Expressions Character classes



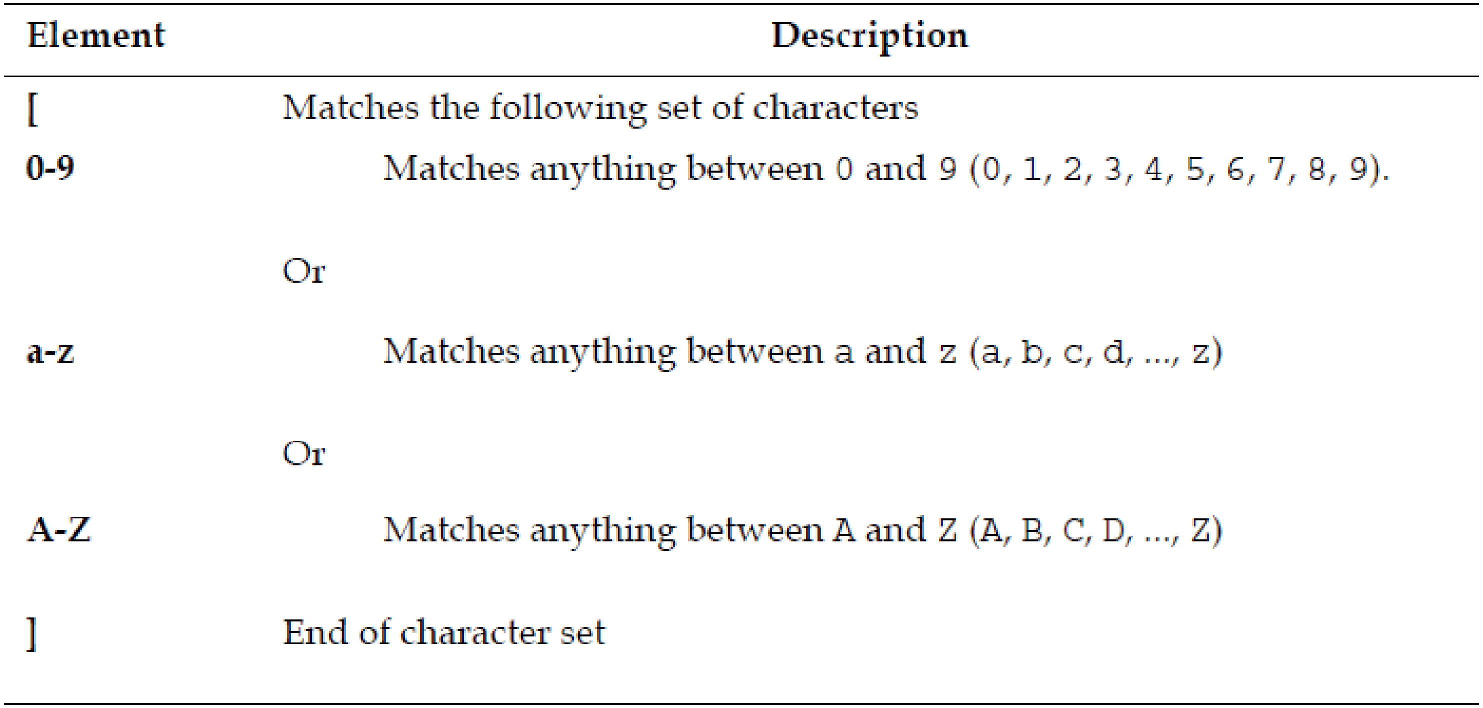
Searching using a character class

- It is possible to also use the range of a character.
- This is done by leveraging the hyphen symbol (-) between two related characters; for example, to match any lowercase letter we can use [a-z].
- Likewise, to match any single digit we can define the character set [0-9].

Regular Expression

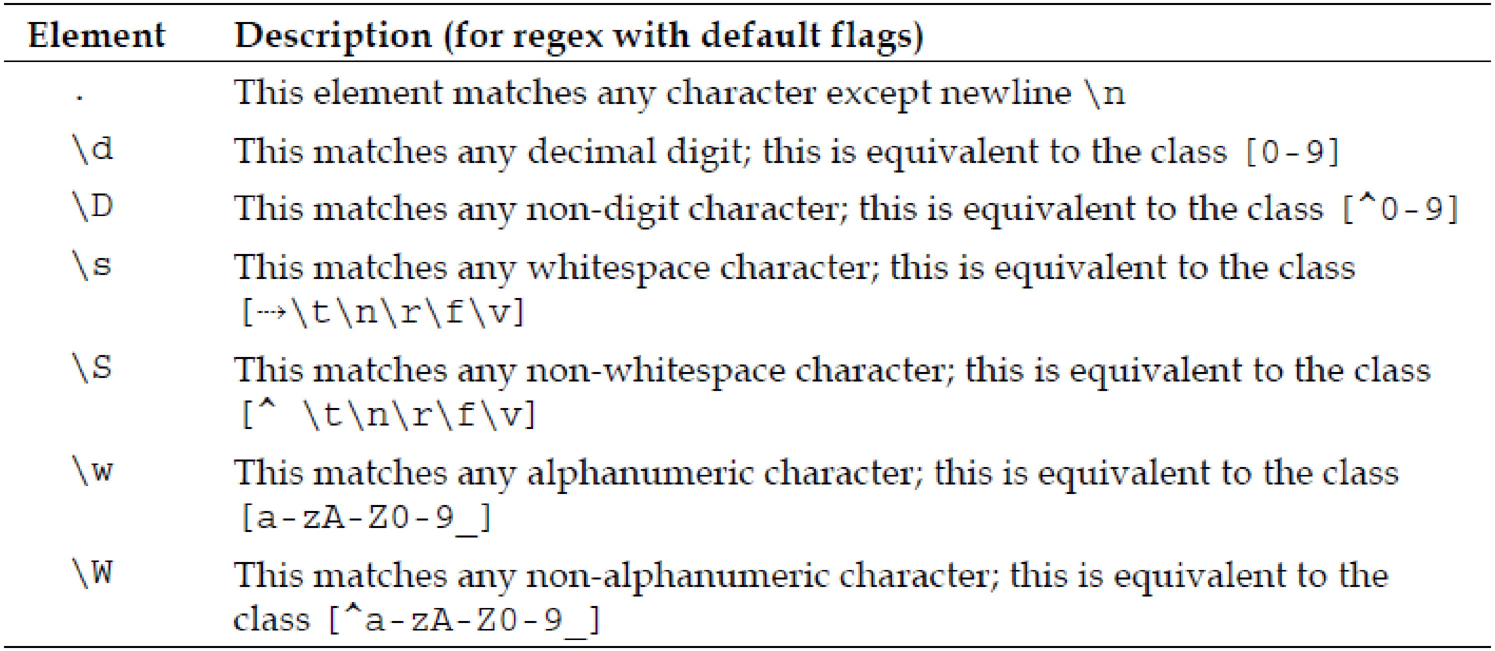
The character classes' ranges can be combined to be able to match a character against many ranges by just putting one range after the other—no special separation is required.

For instance, if we want to match any lowercase or uppercase alphanumeric character, we can use [0-9a-zA-Z] (see next table for a more detailed explanation). This can be alternatively written using the union mechanism: [0-9[a-z[A-Z]]].



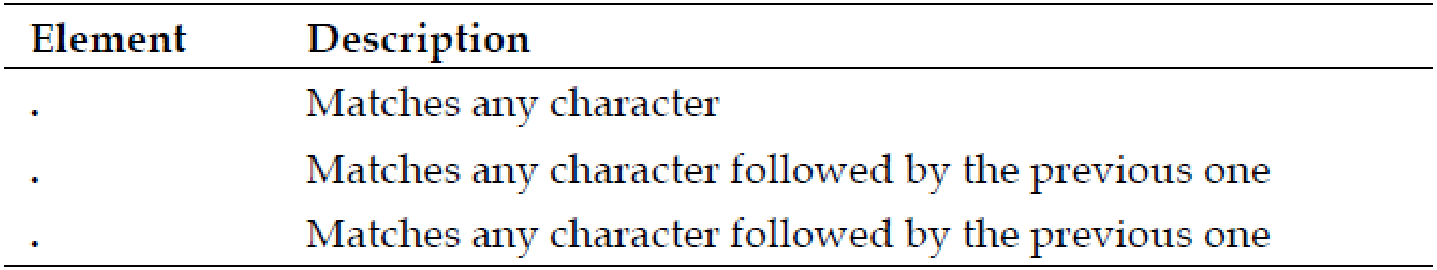
Element	Description
[	Matches the following set of characters
0-9	Matches anything between 0 and 9 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).
	Or
a-z	Matches anything between a and z (a, b, c, d, ..., z)
	Or
A-Z	Matches anything between A and Z (A, B, C, D, ..., Z)
]	End of character set

Predefined character classes

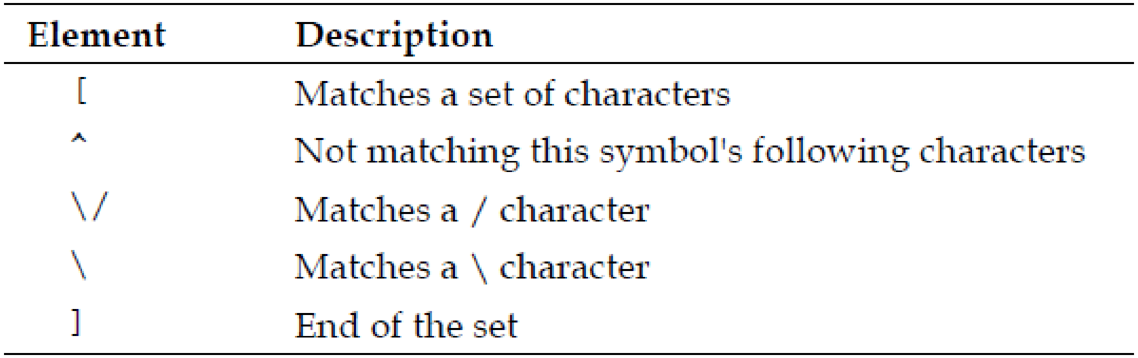


Element	Description (for regex with default flags)
.	This element matches any character except newline \n
\d	This matches any decimal digit; this is equivalent to the class [0-9]
\D	This matches any non-digit character; this is equivalent to the class [^0-9]
\s	This matches any whitespace character; this is equivalent to the class [⇢\t\n\r\f\v]
\S	This matches any non-whitespace character; this is equivalent to the class [^ \t\n\r\f\v]
\w	This matches any alphanumeric character; this is equivalent to the class [a-zA-Z0-9_]
\W	This matches any non-alphanumeric character; this is equivalent to the class [^a-zA-Z0-9_]

Predefined character classes

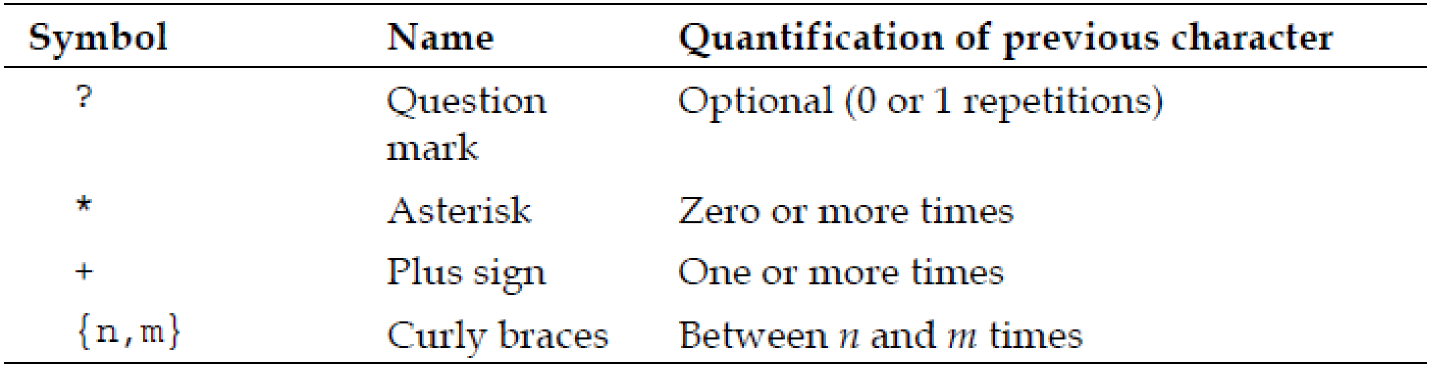


Element	Description
.	Matches any character
.	Matches any character followed by the previous one
.	Matches any character followed by the previous one



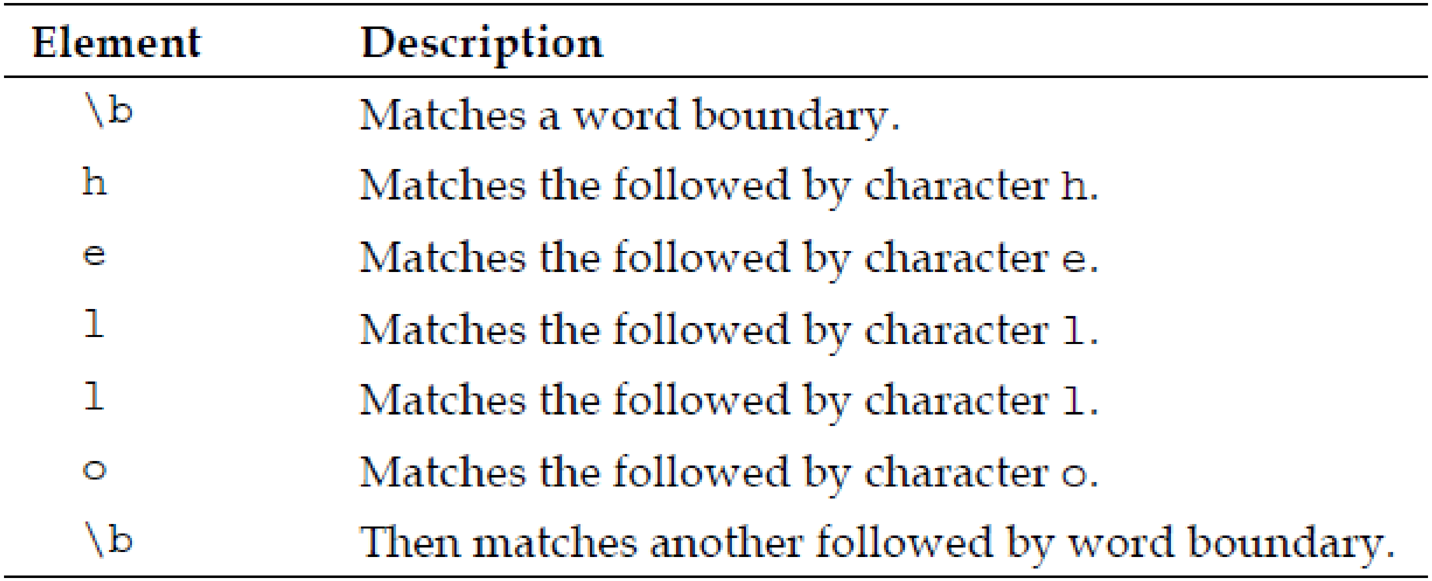
Element	Description
[	Matches a set of characters
^	Not matching this symbol's following characters
\/	Matches a / character
\	Matches a \ character
]	End of the set

Quantifiers



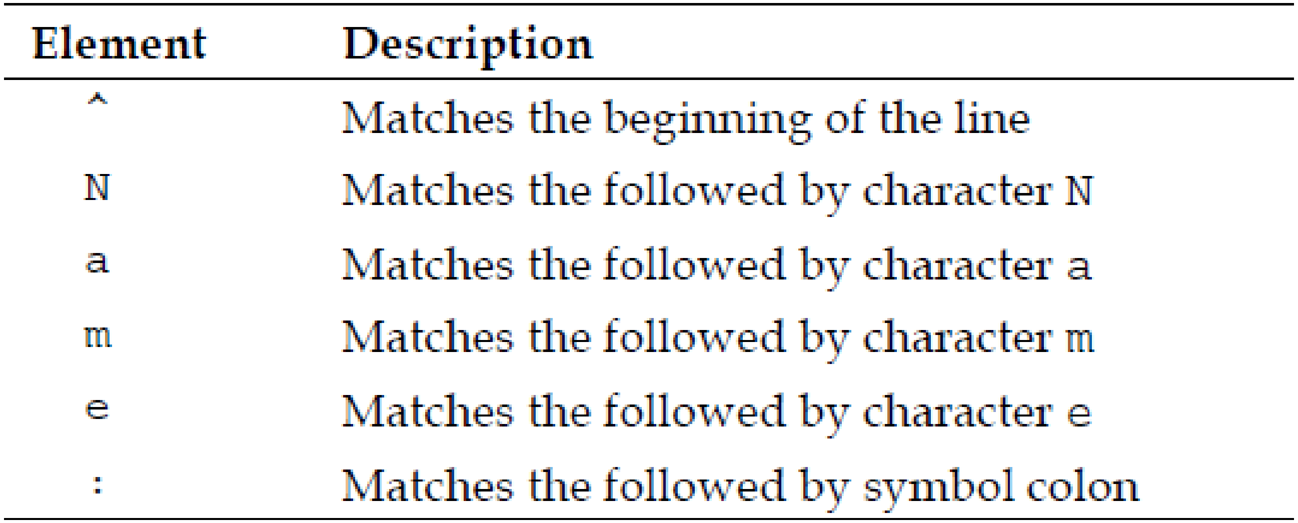
Symbol	Name	Quantification of previous character
`?`	Question mark	Optional (0 or 1 repetitions)
`*`	Asterisk	Zero or more times
`+`	Plus sign	One or more times
`{n,m}`	Curly braces	Between *n* and *m* times

Boundary Matchers



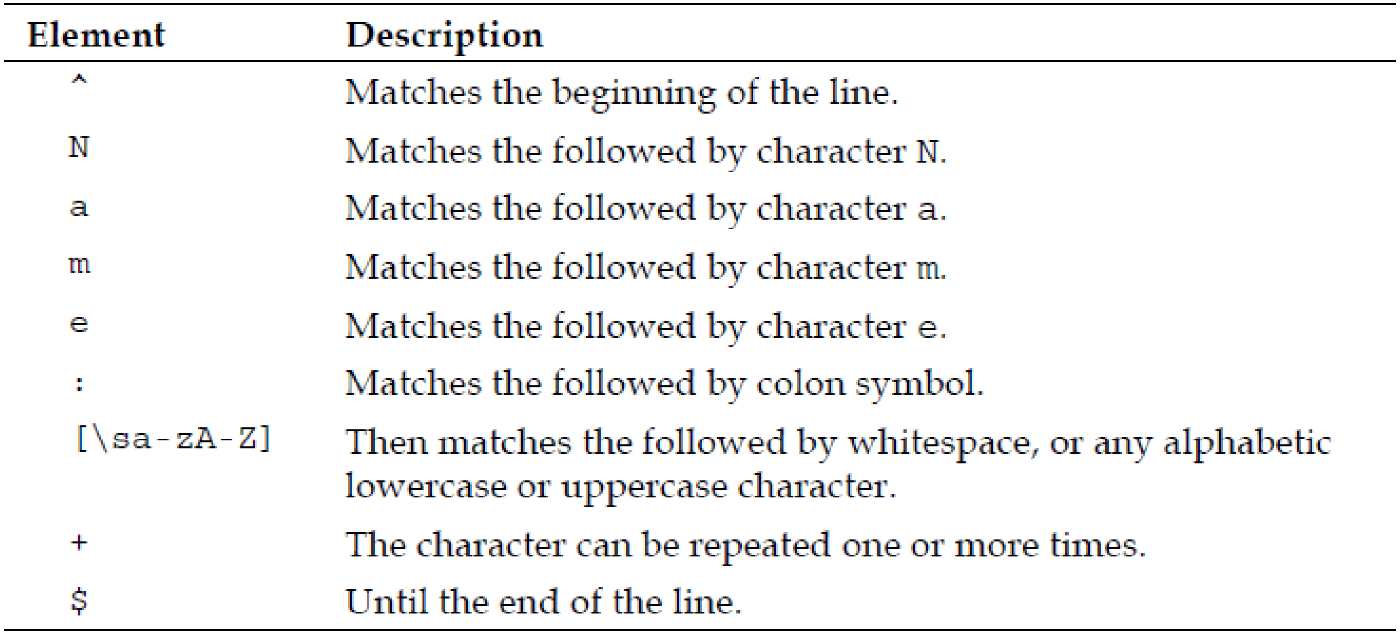
Element	Description
\b	Matches a word boundary.
h	Matches the followed by character h.
e	Matches the followed by character e.
l	Matches the followed by character l.
l	Matches the followed by character l.
o	Matches the followed by character o.
\b	Then matches another followed by word boundary.

Boundary Matchers



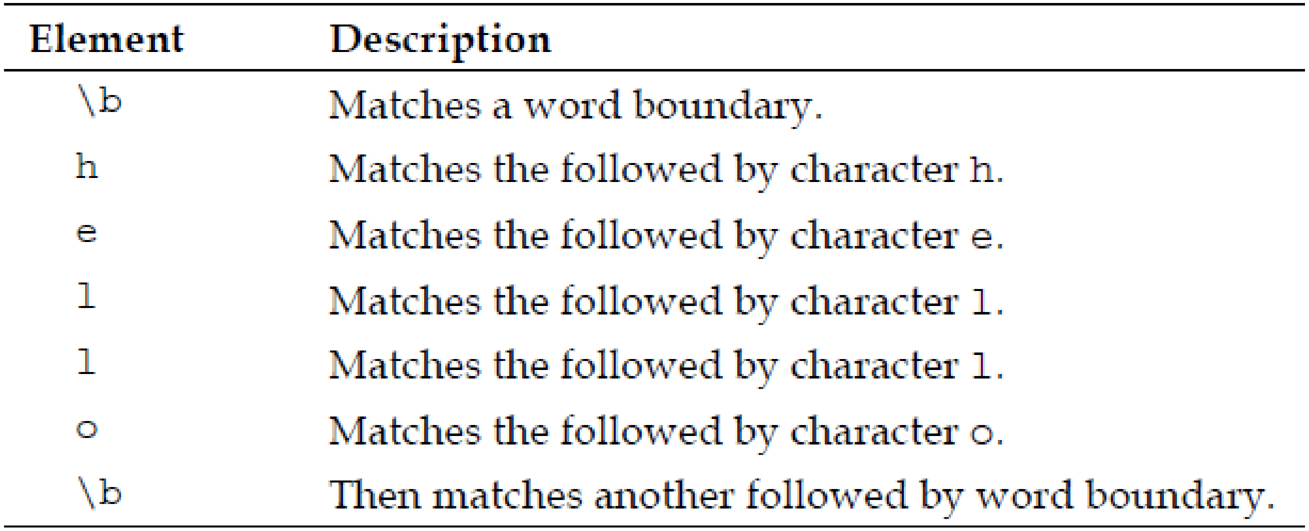
Element	Description
^	Matches the beginning of the line
N	Matches the followed by character N
a	Matches the followed by character a
m	Matches the followed by character m
e	Matches the followed by character e
:	Matches the followed by symbol colon

Boundary Matchers



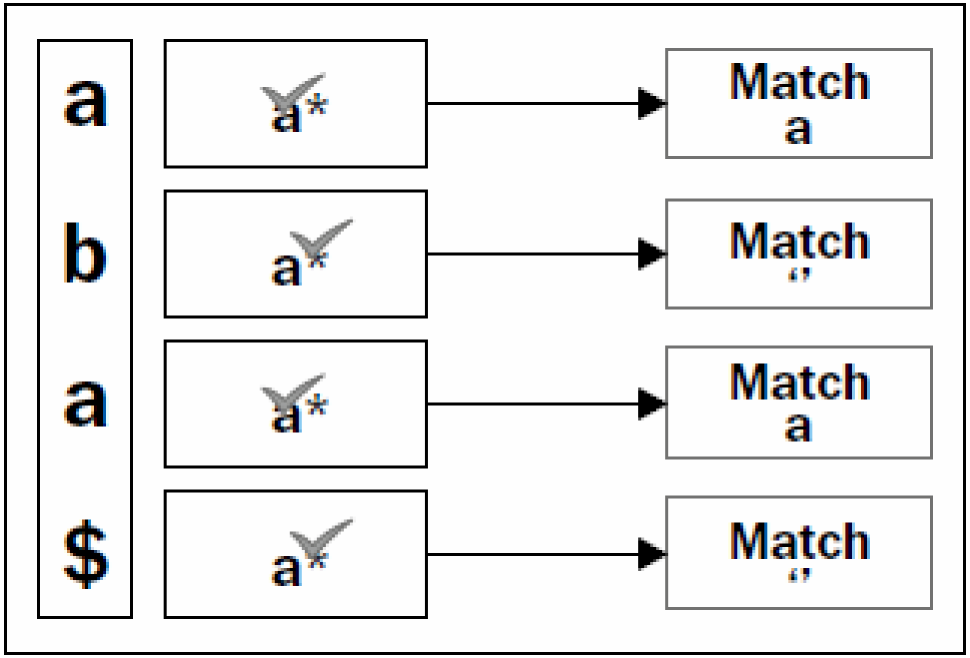
Element	Description
^	Matches the beginning of the line.
N	Matches the followed by character N.
a	Matches the followed by character a.
m	Matches the followed by character m.
e	Matches the followed by character e.
:	Matches the followed by colon symbol.
[\sa-zA-Z]	Then matches the followed by whitespace, or any alphabetic lowercase or uppercase character.
+	The character can be repeated one or more times.
$	Until the end of the line.

Boundary Matchers



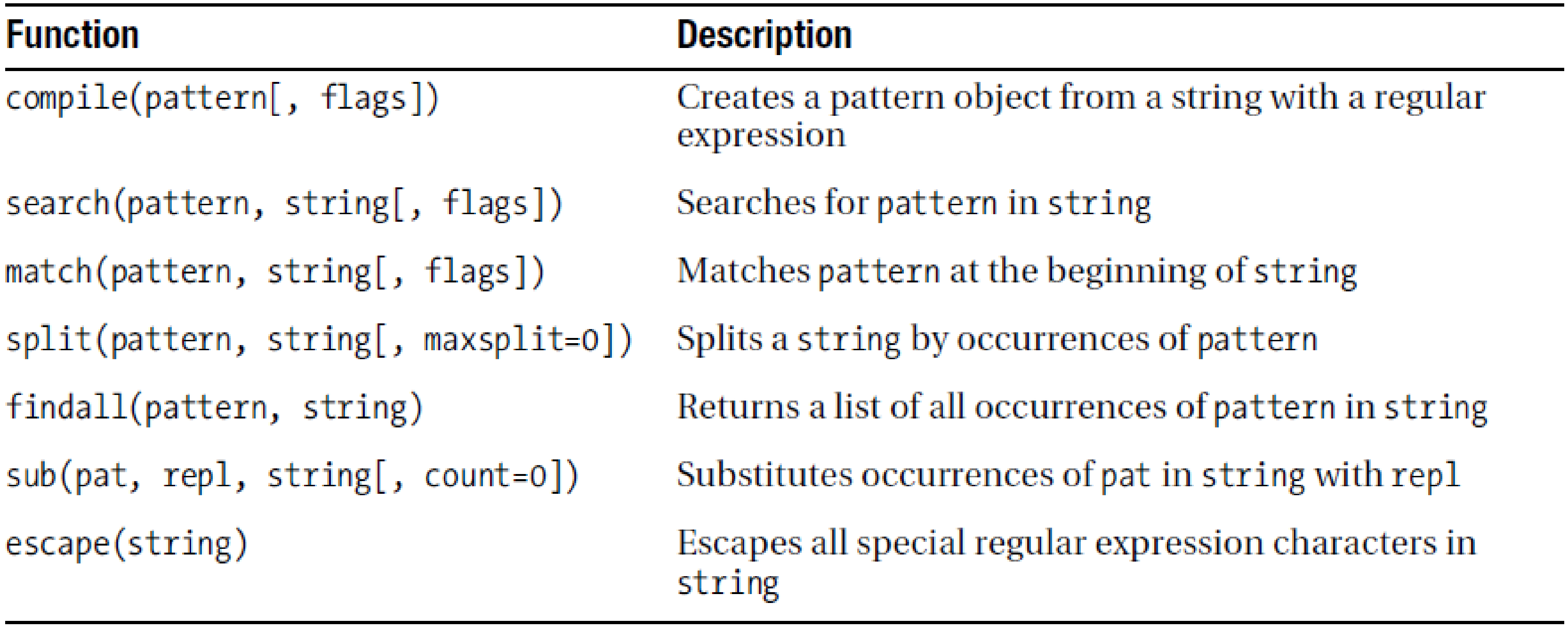
Element	Description
`\b`	Matches a word boundary.
`h`	Matches the followed by character `h`.
`e`	Matches the followed by character `e`.
`l`	Matches the followed by character `l`.
`l`	Matches the followed by character `l`.
`o`	Matches the followed by character `o`.
`\b`	Then matches another followed by word boundary.

Boundary Matchers



findall matching process

Functions



Function	Description
compile(pattern[, flags])	Creates a pattern object from a string with a regular expression
search(pattern, string[, flags])	Searches for pattern in string
match(pattern, string[, flags])	Matches pattern at the beginning of string
split(pattern, string[, maxsplit=0])	Splits a string by occurrences of pattern
findall(pattern, string)	Returns a list of all occurrences of pattern in string
sub(pat, repl, string[, count=0])	Substitutes occurrences of pat in string with repl
escape(string)	Escapes all special regular expression characters in string

Regular Expressions

[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
$	Ends with	"world$"
*	Zero or more occurrences	"aix*"
+	One or more occurrences	"aix+"
{}	Excactly the specified number of occurrences	"al{2}"
|	Either or	"falls|stays"
()	Capture and group	

Regular Expressions

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

Regular Expressions

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) are present
[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., |, (), $,{} has no special meaning, so [+] means: return a match for any +character in the string

Regular Expression

Expression	String	Matched?
^a...s$	abs	No match
	alias	Match
	abyss	Match
	Alias	No match
	An abacus	No match

Regular Expression Meta Characters

[] - Square brackets

Square brackets specifies a set of characters you wish to match.

Expression	String	Matched?
[abc]	a	1 match
	ac	2 matches
	Hey Jude	No match
	abc de ca	5 matches

Regular Expression Meta Characters

. - Period

A period matches any single character (except newline '\n').

Expression	String	Matched?
..	a	No match
	ac	1 match
	acd	1 match
	acde	2 matches (contains 4 characters)

Regular Expression Meta Characters

^ - Caret

The caret symbol ^ is used to check if a string starts with a certain character.

Expression	String	Matched?
^a	a	1 match
	abc	1 match
	bac	No match
^ab	abc	1 match
	acb	No match (starts with a but not followed by b)

Regular Expression Meta Characters

$ - Dollar

The dollar symbol $ is used to check if a string ends with a certain character..

Expression	String	Matched?
a$	a	1 match
	formula	1 match
	cab	No match

Regular Expression Meta Characters

* - Star

The star symbol * matches zero or more occurrences of the pattern left to it..

Expression	String	Matched?
ma*n	mn	1 match
	man	1 match
	maaan	1 match
	main	No match (a is not followed by n)
	woman	1 match

Regular Expression Meta Characters

+ - Plus

The plus symbol + matches one or more occurrences of the pattern left to it.

Expression	String	Matched?
ma+n	mn	No match (no a character)
	man	1 match
	maaan	1 match
	main	No match (a is not followed by n)
	woman	1 match

Regular Expression Meta Characters

? - Question Mark

The question mark symbol ? matches zero or one occurrence of the pattern left to it.

Expression	String	Matched?
ma?n	mn	1 match
	man	1 match
	maaan	No match (more than one a character)
	main	No match (a is not followed by n)
	woman	1 match

Regular Expression Meta Characters

{} - Braces

Consider this code: {n,m}. This means at least n, and at most m repetitions of the pattern left to it.

Expression	String	Matched?
a{2,3}	abc dat	No match
	abc daat	1 match (at daat)
	aabc daaat	2 matches (at aabc and daaat)
	aabc daaaat	2 matches (at aabc and daaaat)

Regular Expression Meta Characters

Special Sequences

Special sequences make commonly used patterns easier to write. Here's a list of special sequences:

\A - Matches if the specified characters are at the start of a string.

Expression	String	Matched?
\Athe	the sun	Match
	In the sun	No match

Regular Expression Meta Characters

\b - Matches if the specified characters are at the beginning or end of a word.

Expression	String	Matched?
\bfoo	football	Match
	a football	Match
	afootball	No match
foo\b	the foo	Match
	the afoo test	Match
	the afootest	No match

Regular Expression Meta Characters

\B - Opposite of \b. Matches if the specified characters are not at the beginning or end of a word.

Expression	String	Matched?
\Bfoo	football	No match
	a football	No match
	afootball	Match
foo\B	the foo	No match
	the afoo test	No match
	the afootest	Match

Regular Expression Meta Characters

\d - Matches any decimal digit. Equivalent to [0-9]

Expression	String	Matched?
\d	12abc3	3 matches (at 12abc3)
	Python	No match

Regular Expression Meta Characters

\D - Matches any non-decimal digit. Equivalent to [^0-9]

Expression	String	Matched?
\D	1ab34"50	3 matches (at 1ab34"50)
	1345	No match

Regular Expression Meta Characters

\s - Matches where a string contains any whitespace character. Equivalent to [\t\n\r\f\v].

Expression	String	Matched?
\s	Python RegEx	1 match
	PythonRegEx	No match

Regular Expression Meta Characters

\w - Matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9_]. By the way, underscore _ is also considered an alphanumeric character.

Expression	String	Matched?
\w	12&": ;c	3 matches (at 12&": ;c)
	%"> !	No match

Regular Expression Meta Characters

\Z - Matches if the specified characters are at the end of a string.

Expression	String	Matched?
\ZPython	I like Python	1 match
	I like Python	No match
	Python is fun.	No match

Regular Expression

re.findall()
The re.findall() method returns a list of strings containing all matches.

```
string = 'hello 12 hi 89. Howdy 34'
pattern = '\d+'

result = re.findall(pattern, string)
print(result)

# Output: ['12', '89', '34']
```

Regular Expression

re.split()
The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

```
string = 'Twelve:12 Eighty nine:89.'
pattern = '\d+'

result = re.split(pattern, string)
print(result)

# Output: ['Twelve:', ' Eighty nine:', '.']
```

Regular Expression

re.split()
The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

```
string = 'Twelve:12 Eighty nine:89.'
pattern = '\d+'

result = re.split(pattern, string)
print(result)

# Output: ['Twelve:', ' Eighty nine:', '.']
```

Regular Expression

re.split()

The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

```
string = 'Twelve:12 Eighty nine:89 Nine:9.'
pattern = '\d+'

# maxsplit = 1
# split only at the first occurrence
result = re.split(pattern, string, 1)
print(result)

# Output: ['Twelve:', ' Eighty nine:89 Nine:9.']
```

Regular Expression

re.sub()
The syntax of re.sub() is:
re.sub(pattern, replace, string)
The method returns a string where matched occurrences are replaced with the content of replace variable.

```
 multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'

# empty string
replace = ''

new_string = re.sub(pattern, replace, string)
print(new_string)

# Output: abc12de23f456
```

Regular Expression

re.sub()
The syntax of re.sub() is:
re.sub(pattern, replace, string)
The method returns a string where matched occurrences are replaced with the content of replace variable.

```
# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'
replace = ''

new_string = re.sub(r'\s+', replace, string, 1)
print(new_string)

# Output:
# abc12de 23
# f45 6
```

Regular Expression

re.subn()
The re.subn() is similar to re.sub() expect it returns a tuple of 2 items containing the new string and the number of substitutions made..

```
# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'

# empty string
replace = ‘ ‘

new_string = re.subn(pattern, replace, string)
print(new_string)

# Output: ('abc12de23f456', 4)
```

Regular Expression

re.search()
The re.search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.

If the search is successful, re.search() returns a match object; if not, it returns None.

```
match = re.search(pattern, str)
```

Regular Expression

```
string = "Python is fun"

# check if 'Python' is at the beginning
match = re.search('\APython', string)

if match:
  print("pattern found inside the string")
else:
  print("pattern not found")

# Output: pattern found inside the string
```

Regular Expression

Match object
You can get methods and attributes of a match object using dir() function.

Some of the commonly used methods and attributes of match objects are:
match.group()
The group() method returns the part of the string where there is a match.

Regular Expression

```
string = '39801 356, 2102 1111'

# Three digit number followed by space followed by two digit number
pattern = '(\d{3}) (\d{2})'

# match variable contains a Match object.
match = re.search(pattern, string)

if match:
  print(match.group())
else:
  print("pattern not found")

# Output: 801 35
```

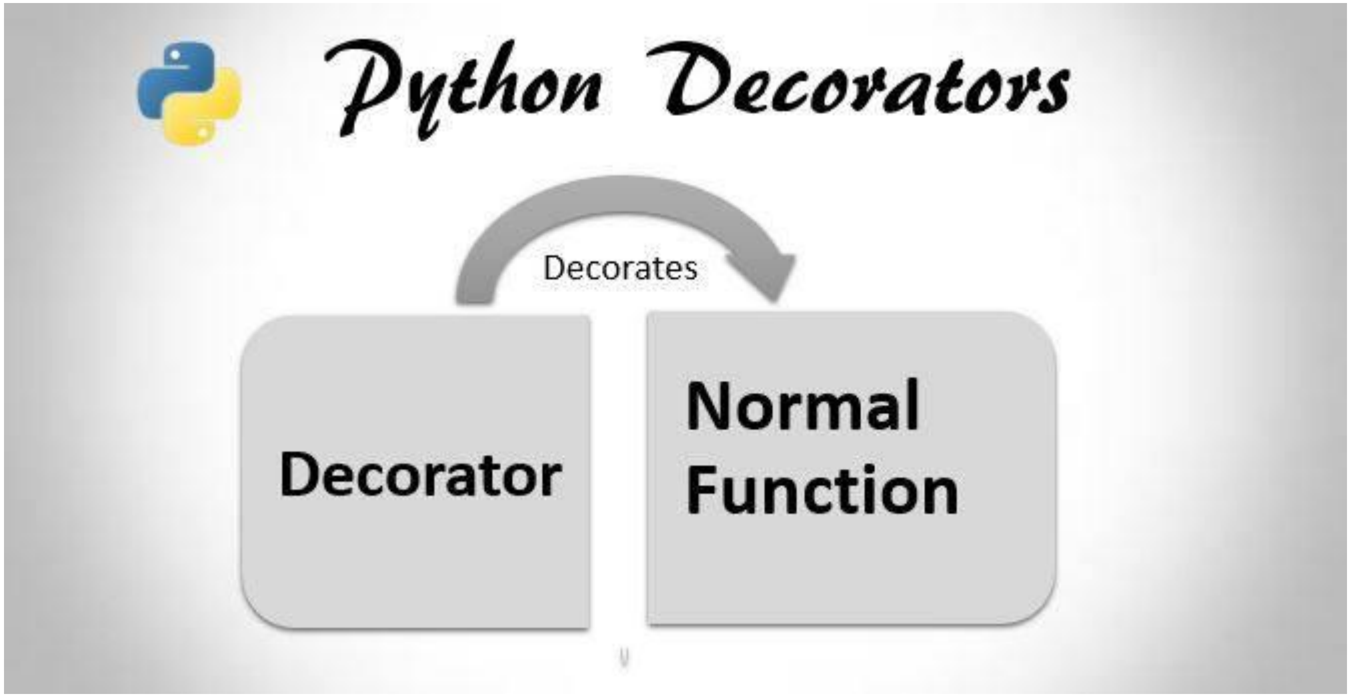
Decorators

- Decorators allow you to inject or modify code in functions or classes.
- Sounds a bit like Aspect-Oriented Programming (AOP) in Java, doesn’t it? Except that it’s both much simpler and (as a result) much more powerful.
- For example, suppose you’d like to do something at the entry and exit points of a function.
- A decorator in Python is any callable Python object that is used to modify a function or a class.

Decorators

we have two different kinds of decorators in Python:

- Function decorators
- Class decorators



Decorators

Function Decorators

A function decorator is applied to a function definition by placing it on the line before that function definition begins.

```
@myDecorator

def aFunction():

print("inside aFunction")
```

When the compiler passes over this code, aFunction() is compiled and the resulting function object is passed to the myDecorator code, which does something to produce a function-like object that is then substituted for the original aFunction().

Decorators

The only constraint upon the object returned by the decorator is that it can be used as a function – which

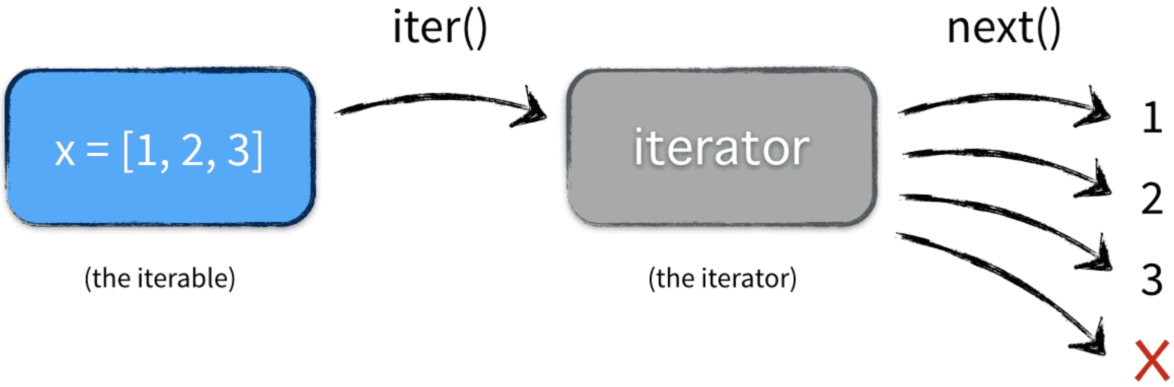
basically means it must be callable. Thus, any classes we use as decorators must implement __call__.

Iterators

- Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but hidden in plain sight.
- Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.
- Technically speaking, Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.
- An object is called iterable if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables.
- The iter() function (which in turn calls the __iter__() method) returns an iterator from them.

Iterators

- Iterators have several advantages:
- Cleaner code
- Iterators can work with infinite sequences
- Iterators save resources
- Python has several built-in objects, which implement the iterator protocol. For example lists, tuples, strings, dictionaries or files



Generators

- Generator is a special routine that can be used to control the iteration behaviour of a loop.
- A generator is similar to a function returning an array.
- A generator has parameters, it can be called and it generates a sequence of numbers.
- But unlike functions, which return a whole array, a generator yields one value at a time. This requires less memory.

Generators

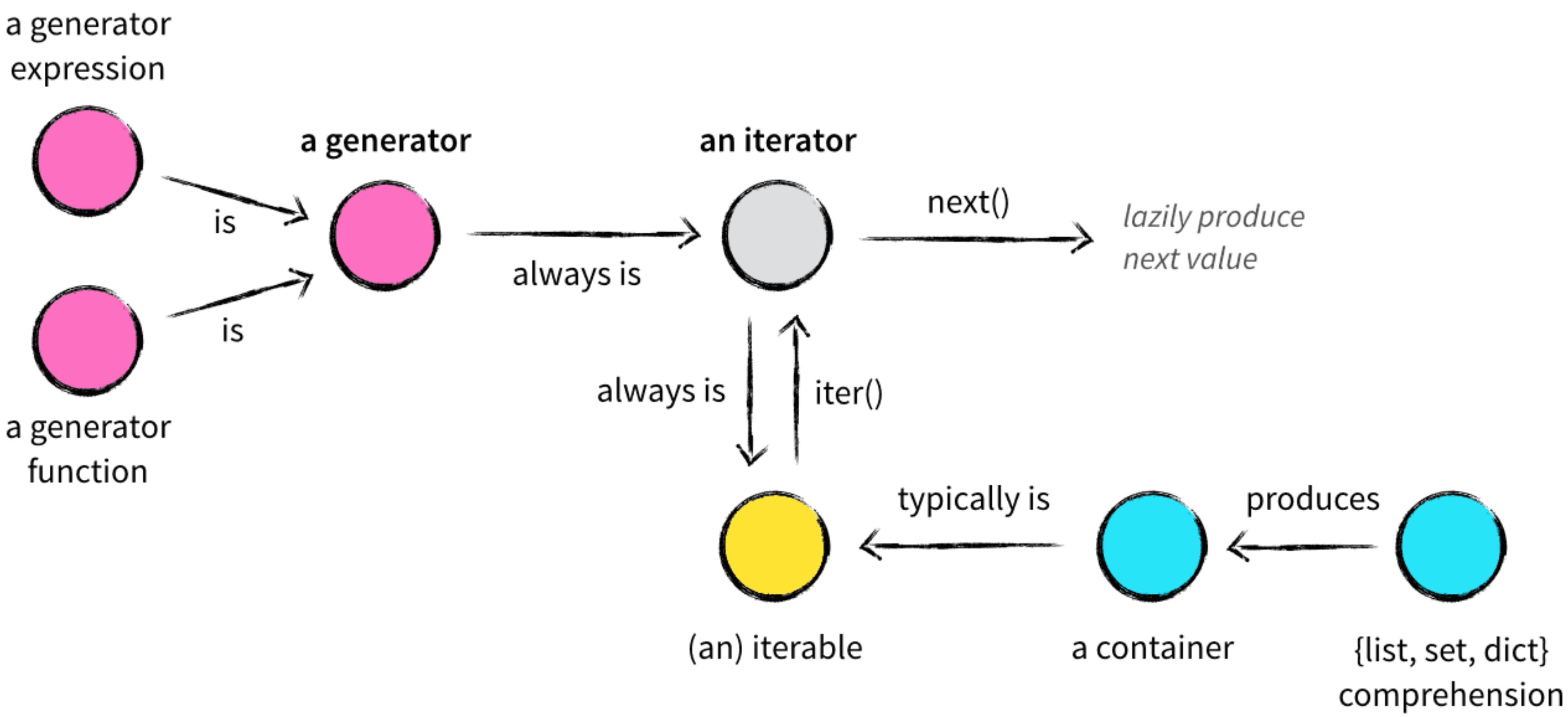
- Generators in Python:

- Are defined with the def keyword
- Use the yield keyword
- May use several yield keywords
- Return an iterator

Generators vs Iterators

- While creating a Python Generator, we use function. But to create an iterator in Python we use iter() and next() functions.
- A generator use ‘yield’ keyword which Python iterator doesn't.
- Python Generator saves the state of local variable every time yield pauses the loop in python. Iterator dose not use local variables. It just needs iterable to iterate on.
- A generator can have any number of ‘yield’ statement.
- You cam implement your own iterator using python class. A generator does not need a class.
- You can either use a Python function or a comprehension to write Python Generator. For iterator you have to use iter() and next() functions.
- We can write fast and compact code with Python Generator. This is an advantage over python iterators. They are also simpler to code than iterators.

Generators vs Iterators



Collections Framework

- Collections in Python are containers that are used to store collections of data, for example, list, dict, set, tuple etc.
- These are built-in collections.
- Several modules have been developed that provide additional data structures to store collections of data.
- One such module is the Python collections module.

Collections Module

- Counter
- defaultdict
- OrderedDict
- deque
- ChainMap
- namedtuple()

Collections Module

- The Counter
- Counter is a subclass of dictionary object.
- The Counter() function in collections module takes an iterable or a mapping as the argument and returns a Dictionary.
- In this dictionary, a key is an element in the iterable or the mapping and value is the number of times that element exists in the iterable or the mapping.
- You have to import the Counter class before you can create a counter instance.

Collections Module

- The Counter
- Counter is a subclass of dictionary object.
- The Counter() function in collections module takes an iterable or a mapping as the argument and returns a Dictionary.
- In this dictionary, a key is an element in the iterable or the mapping and value is the number of times that element exists in the iterable or the mapping.
- You have to import the Counter class before you can create a counter instance.

Collections Module

- The defaultdict
- The defaultdict works exactly like a python dictionary, except for it does not throw KeyError when you try to access a non-existent key.
- Instead, it initializes the key with the element of the data type that you pass as an argument at the creation of defaultdict. The data type is called default_factory.

Collections Module

- The OrderedDict
- OrderedDict is a dictionary where keys maintain the order in which they are inserted, which means if you change the value of a key later, it will not change the position of the key.

- The deque (Double Ended Queue)
- The deque is a list optimized for inserting and removing items.

Collections Module

- The ChainMap
- ChainMap is used to combine several dictionaries or mappings. It returns a list of dictionaries.

- The namedtuple()
- The namedtuple() returns a tuple with names for each position in the tuple.
- One of the biggest problems with ordinary tuples is that you have to remember the index of each field of a tuple object.
- This is obviously difficult. The namedtuple was introduced to solve this problem.

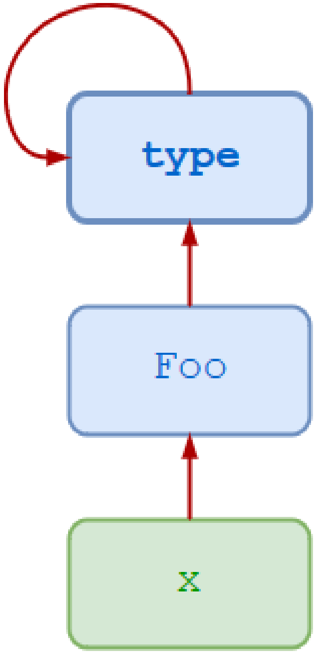
Meta class

- A metaclass is a class whose instances are classes. Like an "ordinary" class defines the behavior of the instances of the class, a metaclass defines the behavior of classes and their instances.
- Metaclasses are not supported by every object oriented programming language. Those programming language, which support metaclasses, considerably vary in way the implement them. Python is supporting them

Meta Class

- There are numerous use cases for metaclasses. Just to name a few:
- logging and profiling
- interface checking
- registering classes at creation time
- automatically adding new methods
- automatic property creation
- proxies
- automatic resource locking/synchronization.

Meta Class



Dynamic Meta class

- >>> type(3)
- <class 'int'>

- >>> type(['foo', 'bar', 'baz'])
- <class 'list'>

- >>> t = (1, 2, 3, 4, 5)
- >>> type(t)
- <class 'tuple'>

Dynamic Meta class

- >>> class Foo:
- ...   pass
- ...
- >>> type(Foo())
- <class '__main__.Foo'>

Dynamic Meta class

- You can also call type() with three arguments—type(<name>, <bases>, <dct>):
- <name> specifies the class name. This becomes the __name__ attribute of the class.
- <bases> specifies a tuple of the base classes from which the class inherits. This becomes the __bases__ attribute of the class.
- <dct> specifies a namespace dictionary containing definitions for the class body. This becomes the __dict__ attribute of the class.

Dynamic Meta class

- >>> Bar = type('Bar', (Foo,), dict(attr=100))
- >>> x = Bar()
- >>> x.attr
- 100
- >>> x.__class__
- <class '__main__.Bar'>
- >>> x.__class__.__bases__
- (<class '__main__.Foo'>,)

Threads

- In computing, a process is an instance of a computer program that is being executed.
- Any process has 3 basic components:
- An executable program.
- The associated data needed by the program (variables, work space, buffers, etc.)
- The execution context of the program (State of process)

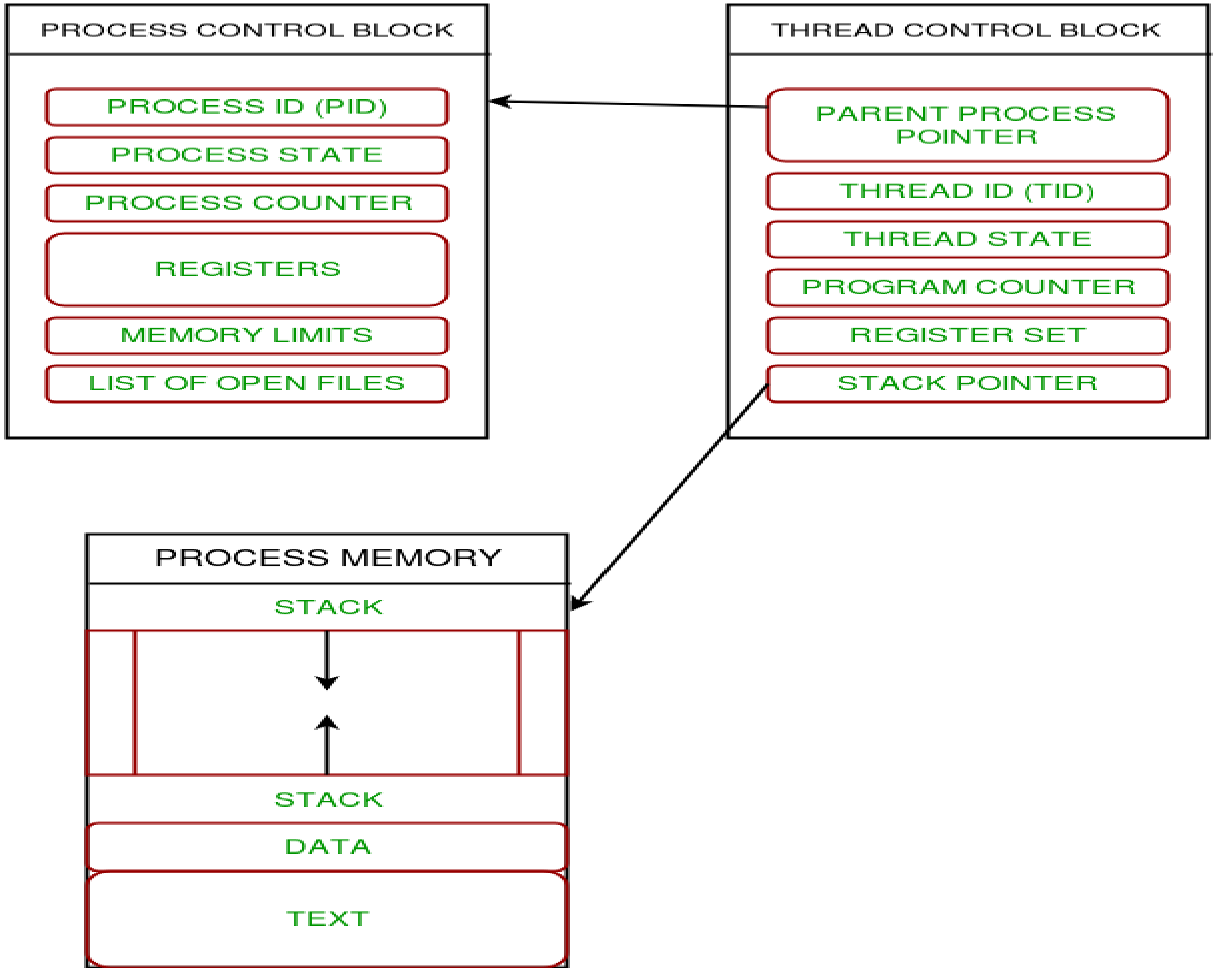
Threads

- A thread is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).
- In simple words, a thread is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process!

Threads

- A thread contains all this information in a Thread Control Block (TCB):
- Thread Identifier: Unique id (TID) is assigned to every new thread
- Stack pointer: Points to thread’s stack in the process. Stack contains the local variables under thread’s scope.
- Program counter: a register which stores the address of the instruction currently being executed by thread.
- Thread state: can be running, ready, waiting, start or done.
- Thread’s register set: registers assigned to thread for computations.
- Parent process Pointer: A pointer to the Process control block (PCB) of the process that the thread lives on.

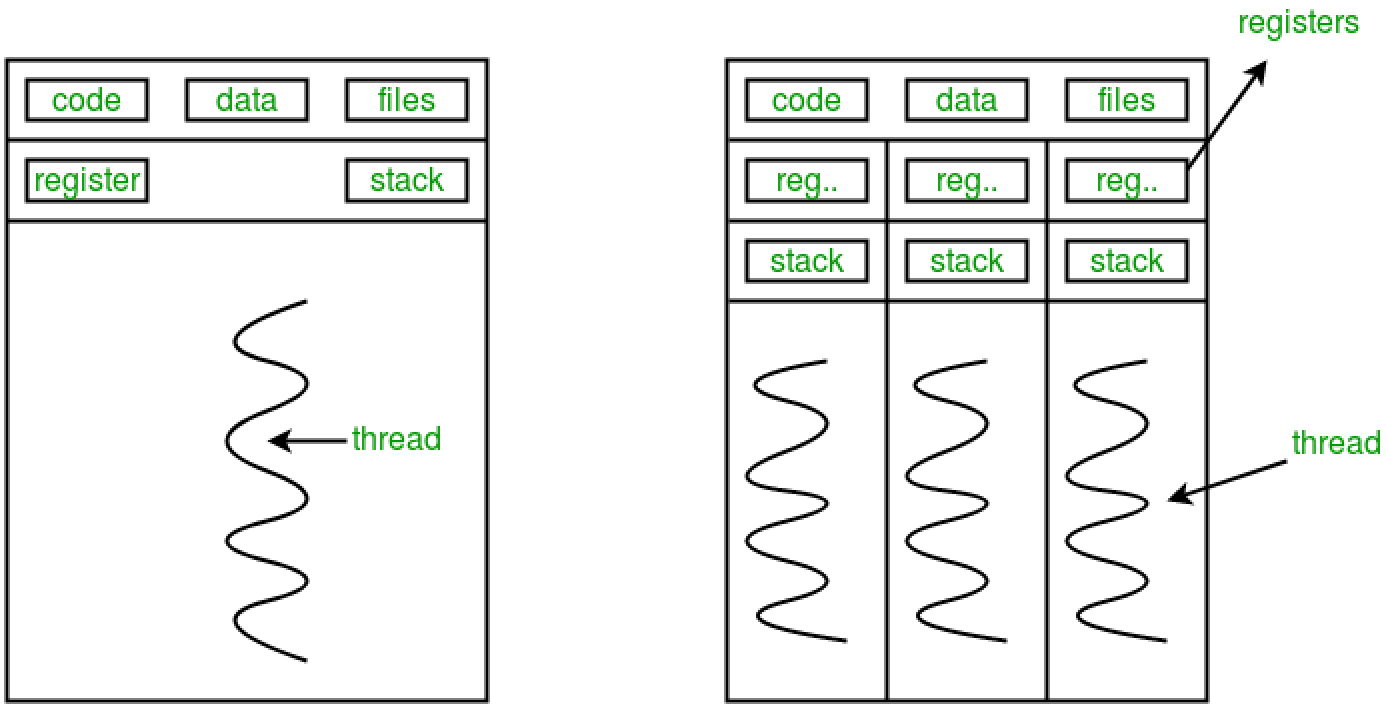
Threads


PROCESS CONTROL BLOCK
PROCESS ID (PID)
PROCESS STATE
PROCESS COUNTER
REGISTERS
MEMORY LIMITS
LIST OF OPEN FILES
THREAD CONTROL BLOCK
PARENT PROCESS POINTER
THREAD ID (TID)
THREAD STATE
PROGRAM COUNTER
REGISTER SET
STACK POINTER
PROCESS MEMORY
STACK
STACK
DATA
TEXT

Threads

- Multiple threads can exist within one process where:
- Each thread contains its own register set and local variables (stored in stack).
- All thread of a process share global variables (stored in heap) and the program code.
- Consider the diagram below to understand how multiple threads exist in memory:

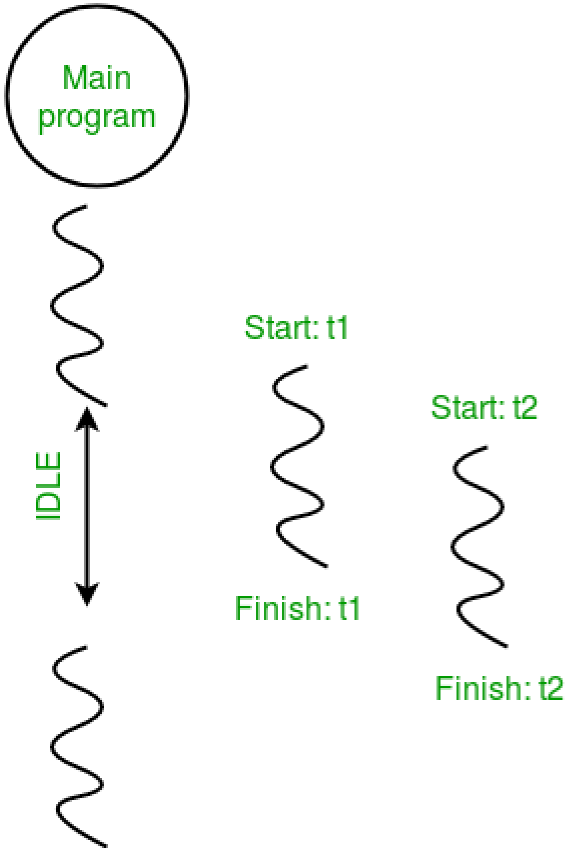
Threads



single-threaded process

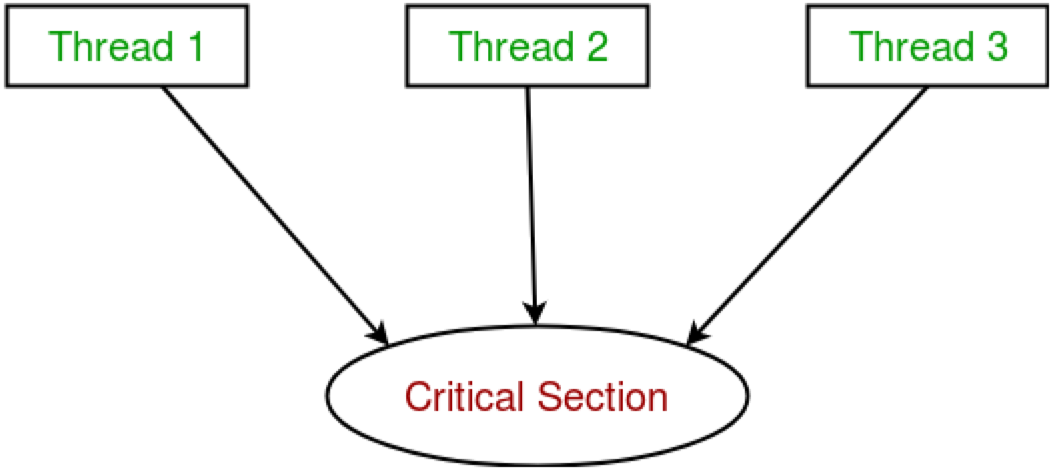
multithreaded process

Threads



Multithreading in Python | Set 2 (Synchronization)

- Thread synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as critical section.



Multithreading

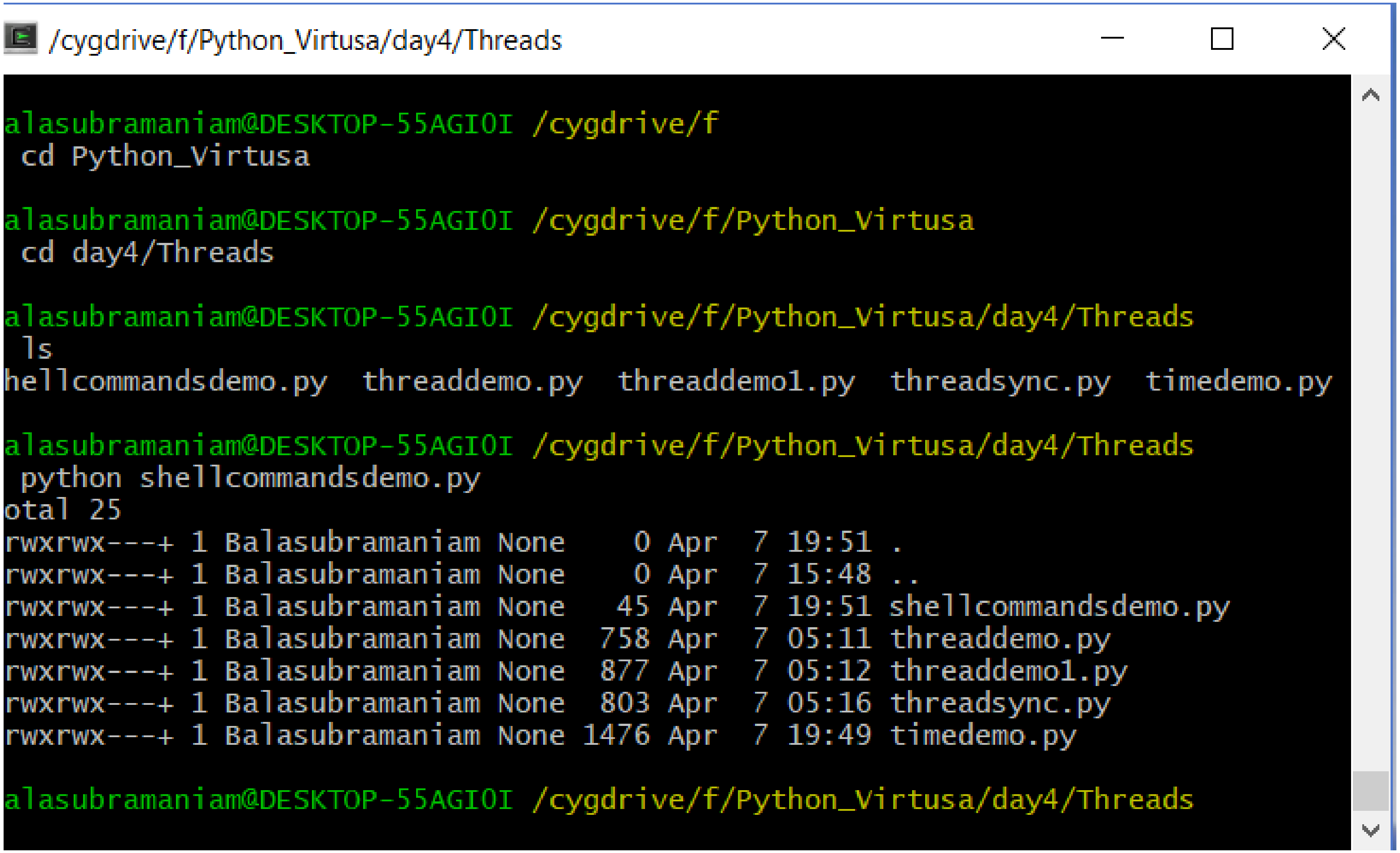
- Advantages:
- It doesn’t block the user. This is because threads are independent of each other.
- Better use of system resources is possible since threads execute tasks parallely.
- Enhanced performance on multi-processor machines.
- Multi-threaded servers and interactive GUIs use multithreading exclusively.

- Disadvantages:
- As number of threads increase, complexity increases.
- Synchronization of shared resources (objects, data) is necessary.
- It is difficult to debug, result is sometimes unpredictable.
- Potential deadlocks which leads to starvation, i.e. some threads may not be served with a bad design
- Constructing and synchronizing threads is CPU/memory intensive.

Time class

- Python has a module named time to handle time-related tasks.
- To use functions defined in the module, we need to import the module first.
- time.time()
- time.ctime()
- time.sleep()
- time.struct_time Class
- time.localtime()
- time.gmtime()
- time.mktime()
- time.asctime()
- time.strftime()
- time.strptime()
-

Shell Commands from Cygwin and Python



```
alasubramaniam@DESKTOP-55AGIOI /cygdrive/f
 cd Python_Virtusa

alasubramaniam@DESKTOP-55AGIOI /cygdrive/f/Python_Virtusa
 cd day4/Threads

alasubramaniam@DESKTOP-55AGIOI /cygdrive/f/Python_Virtusa/day4/Threads
 ls
hellcommandsdemo.py  threaddemo.py  threaddemo1.py  threadsync.py  timedemo.py

alasubramaniam@DESKTOP-55AGIOI /cygdrive/f/Python_Virtusa/day4/Threads
 python shellcommandsdemo.py
otal 25
rwxrwx---+ 1 Balasubramaniam None    0 Apr  7 19:51 .
rwxrwx---+ 1 Balasubramaniam None    0 Apr  7 15:48 ..
rwxrwx---+ 1 Balasubramaniam None   45 Apr  7 19:51 shellcommandsdemo.py
rwxrwx---+ 1 Balasubramaniam None  758 Apr  7 05:11 threaddemo.py
rwxrwx---+ 1 Balasubramaniam None  877 Apr  7 05:12 threaddemo1.py
rwxrwx---+ 1 Balasubramaniam None  803 Apr  7 05:16 threadsync.py
rwxrwx---+ 1 Balasubramaniam None 1476 Apr  7 19:49 timedemo.py

alasubramaniam@DESKTOP-55AGIOI /cygdrive/f/Python_Virtusa/day4/Threads
```

Parallel Processing

- Python offers three possible ways to handle that.
- First, you can execute functions in parallel using the multiprocessing module.
- Second, an alternative to processes are threads.
- Third, you can call external programs using the system() method of the os module, or methods provided by the subprocess module, and collect the results afterwards.

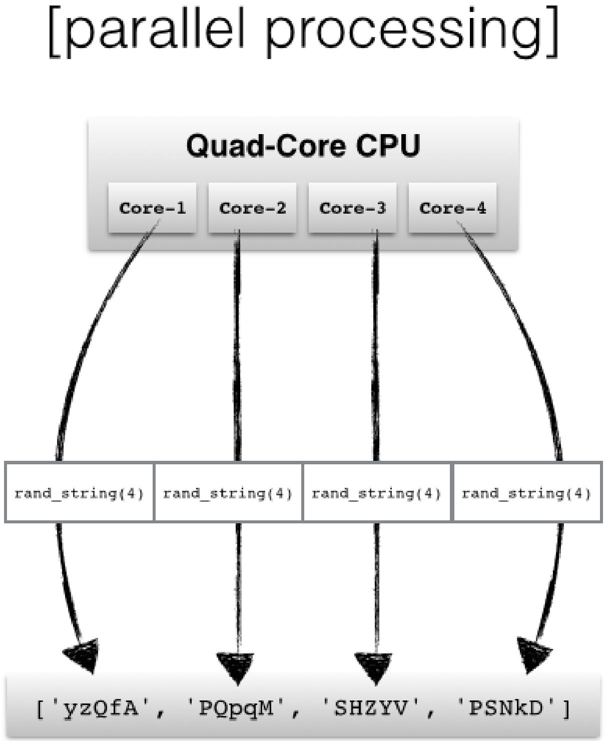
Parallel Processing

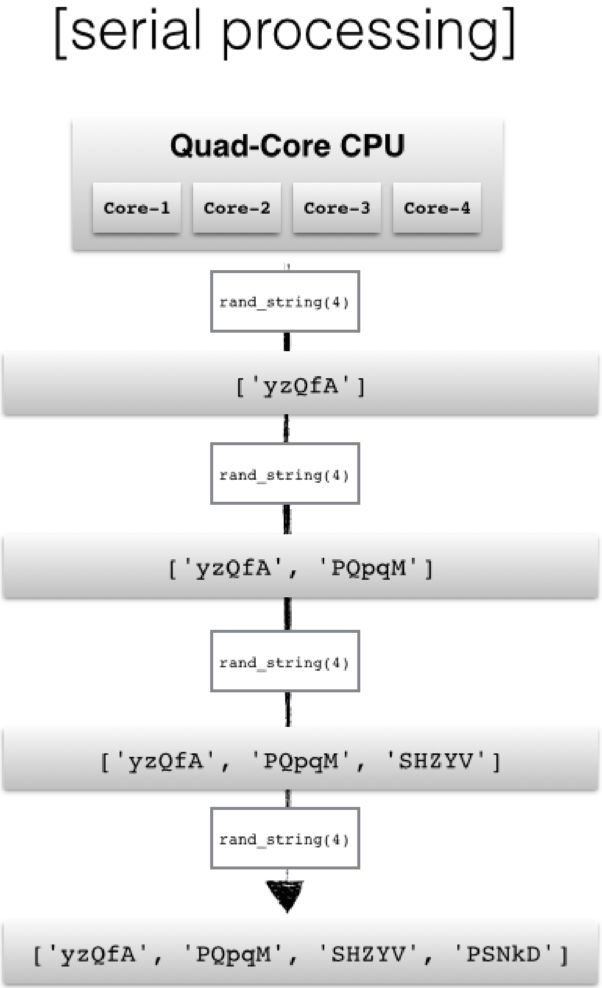
- The multiprocessing module covers a nice selection of methods to handle the parallel execution of routines.
- This includes processes, pools of agents, queues, and pipes.
- What is Synchronous and Asynchronous execution?
- In parallel processing, there are two types of execution: Synchronous and Asynchronous.
- A synchronous execution is one the processes are completed in the same order in which it was started. This is achieved by locking the main program until the respective processes are finished.
- Asynchronous, on the other hand, doesn’t involve locking. As a result, the order of results can get mixed up but usually gets done quicker.

Parallel Processing

- There are 2 main objects in multiprocessing to implement parallel execution of a function: The Pool Class and the Process Class.
- Pool Class
  - Synchronous execution
  - Pool.map() and Pool.starmap()
  - Pool.apply()
- Asynchronous execution
  - Pool.map_async() and Pool.starmap_async()
  - Pool.apply_async())
- Process Class

Parallel Processing





Parallel Processing

Steps to parallelize a function:

- To parallelize a function, you need to execute it in various processors.
- Start with initializing a pool with x number of processors.
- Now, pass that function to any one of the parallelization methods of a pool.
- The apply(), starmap() and map() methods of the multiprocessing.Pool() will allow you to execute functions parallelly.
- Here, both map and apply functions are used to take the functions for parallelization as the main argument.
- The only difference is that map() will only bring one iterable as an argument while the apply() collects args argument which takes the parameters of the parallelized function as an argument.

Re-entrant Locks

- Normal Lock objects cannot be acquired more than once, even by the same thread.
- This can introduce undesirable side-effects if a lock is accessed by more than one function in the same call chain.

Limiting Concurrent Access to Resources

- Sometimes it is useful to allow more than one worker access to a resource at a time, while still limiting the overall number.
- For example, a connection pool might support a fixed number of simultaneous connections, or a network application might support a fixed number of concurrent downloads.
- A Semaphore is one way to manage those connections.

Limiting Concurrent Access to Resources

- A semaphore is based on an internal counter which is decremented each time acquire() is called and incremented each time release() is called.
- If the counter is equal to 0 then acquire() blocks. It is the Python implementation of the Dijkstra semaphore concept: P() and V().
- Using a semaphore makes sense when you want to control access to a resource with limited capacity like a server..

Rentrant Lock

- RLock is the reentrant lock provided by python.
- A reentrant lock or simply RLock allows itself to be locked several times.
- The RLock is released only after equal number release() calls as the acquire() calls made to it.
- Unlike the primitive lock given by the Lock class the RLock can only be released by the thread holding the RLock.

Sockets, IPv4, and Simple Client/Server Programming

- Python's socket module has both class-based and instances-based utilities.
- The difference between a class-based and instance-based method is that the former doesn't need an instance of a socket object.
- For example, in order to print your machine's IP address, you don't need a socket object. Instead, you can just call the socket's class-based methods.
- On the other hand, if you need to send some data to a server application, it is more intuitive that you create a socket object to perform that explicit operation.

Sockets, IPv4, and Simple Client/Server Programming

- The following is a list of those third-party libraries with their download URLs:
- ff ntplib: https://pypi.python.org/pypi/ntplib/
- ff diesel: https://pypi.python.org/pypi/diesel/
- ff nmap: https://pypi.python.org/pypi/python-nmap
- ff scapy: https://pypi.python.org/pypi/scapy
- ff netifaces: https://pypi.python.org/pypi/netifaces/
- ff netaddr: https://pypi.python.org/pypi/netaddr
- ff pyopenssl: https://pypi.python.org/pypi/pyOpenSSL

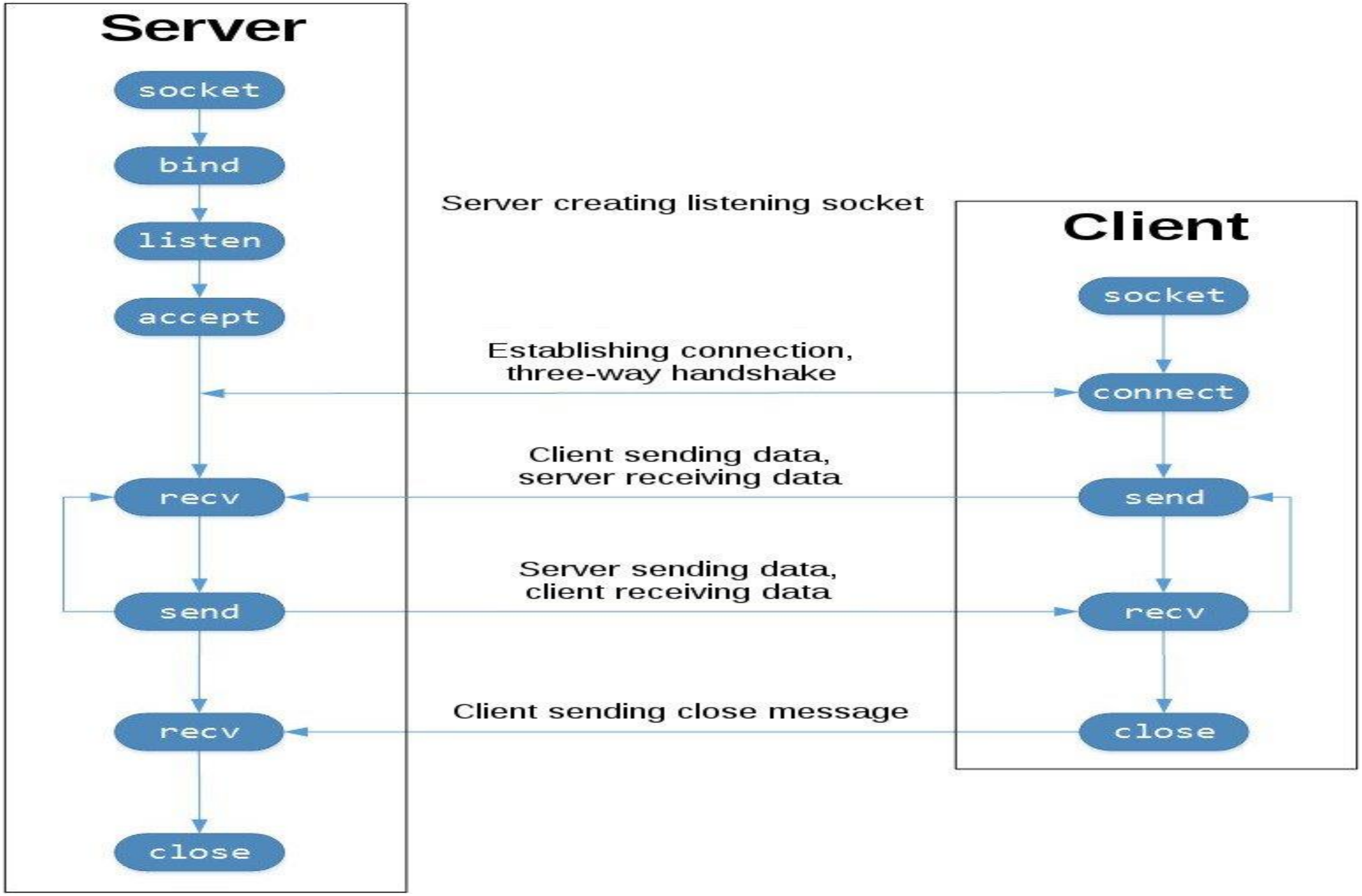
Sockets, IPv4, and Simple Client/Server Programming

- pygeocoder: https://pypi.python.org/pypi/pygocoder
- ff pyyaml: https://pypi.python.org/pypi/PyYAML
- ff requests: https://pypi.python.org/pypi/requests
- ff feedparser: https://pypi.python.org/pypi/feedparser
- ff paramiko: https://pypi.python.org/pypi/paramiko/
- ff fabric: https://pypi.python.org/pypi/Fabric
- ff supervisor: https://pypi.python.org/pypi/supervisor
- ff xmlrpclib: https://pypi.python.org/pypi/xmlrpclib
- ff SOAPpy: https://pypi.python.org/pypi/SOAPpy
- ff bottlenose: https://pypi.python.org/pypi/bottlenose
- ff construct: https://pypi.python.org/pypi/construct

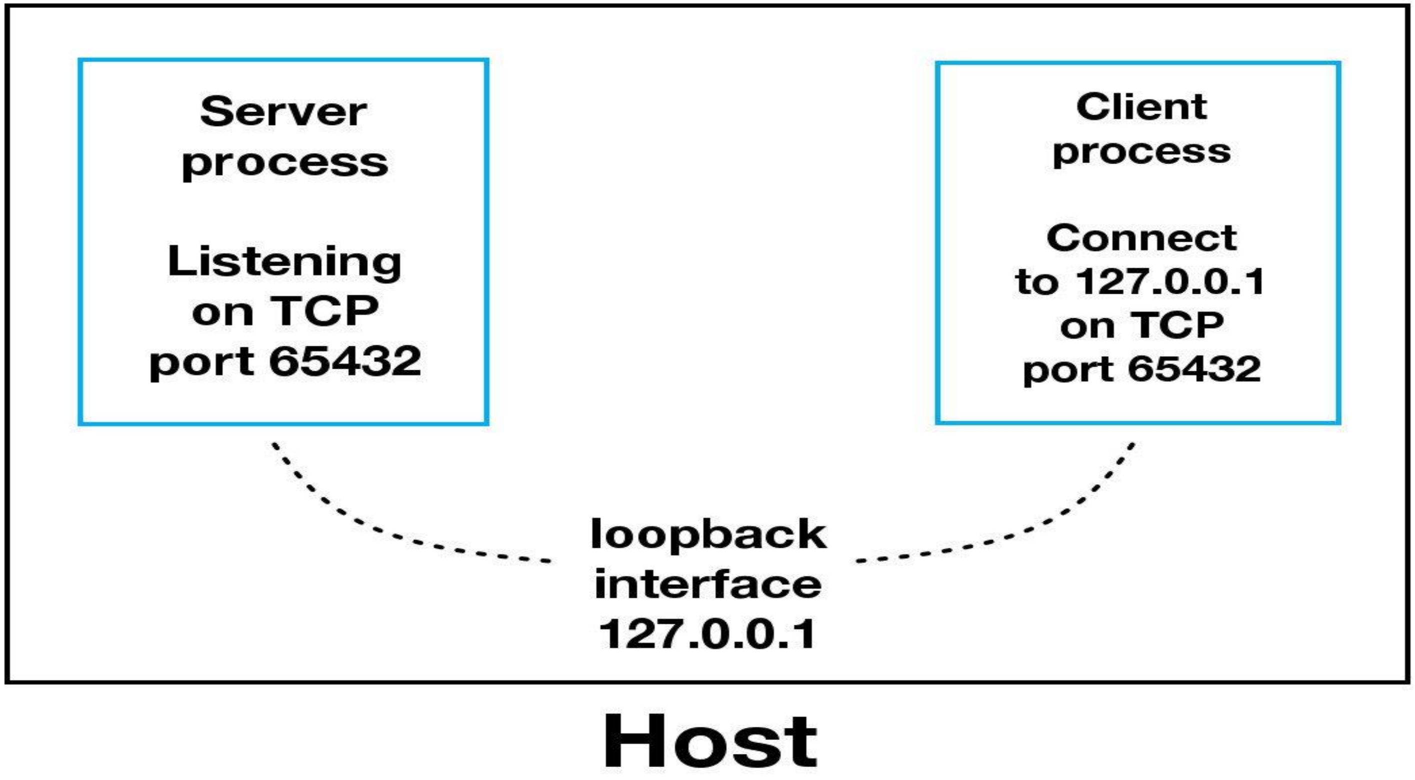
Sockets and Networking

- Socket programming is a way of connecting two nodes on a network to communicate with each other.
- One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection.
- Server forms the listener socket while client reaches out to the server.
- They are the real backbones behind web browsing. In simpler terms there is a server and a client.

Sockets and Networking



Sockets and Networking


Server process
Listening on TCP port 65432
Client process
Connect to 127.0.0.1 on TCP port 65432
loopback interface 127.0.0.1
Host

Signal Handling

- 1 (SIGHUP): terminate a connection, or reload the configuration for daemons
- 2 (SIGINT): interrupt the session from the dialogue station
- 3 (SIGQUIT): terminate the session from the dialogue station
- 4 (SIGILL): illegal instruction was executed
- 5 (SIGTRAP): do a single instruction (trap)
- 6 (SIGABRT): abnormal termination
- 7 (SIGBUS): error on the system bus
- 8 (SIGFPE): floating point error
- 9 (SIGKILL): immmediately terminate the process
- 10 (SIGUSR1): user-defined signal
- 11 (SIGSEGV): segmentation fault due to illegal access of a memory segment
- 12 (SIGUSR2): user-defined signal
- 13 (SIGPIPE): writing into a pipe, and nobody is reading from it
- 14 (SIGALRM): the timer terminated (alarm)
- 15 (SIGTERM): terminate the process in a soft way

Twisted

- The main idea behind Twisted is that it gives us the parallelism of multithreading programming with the ease of reasoning of single threaded programming.

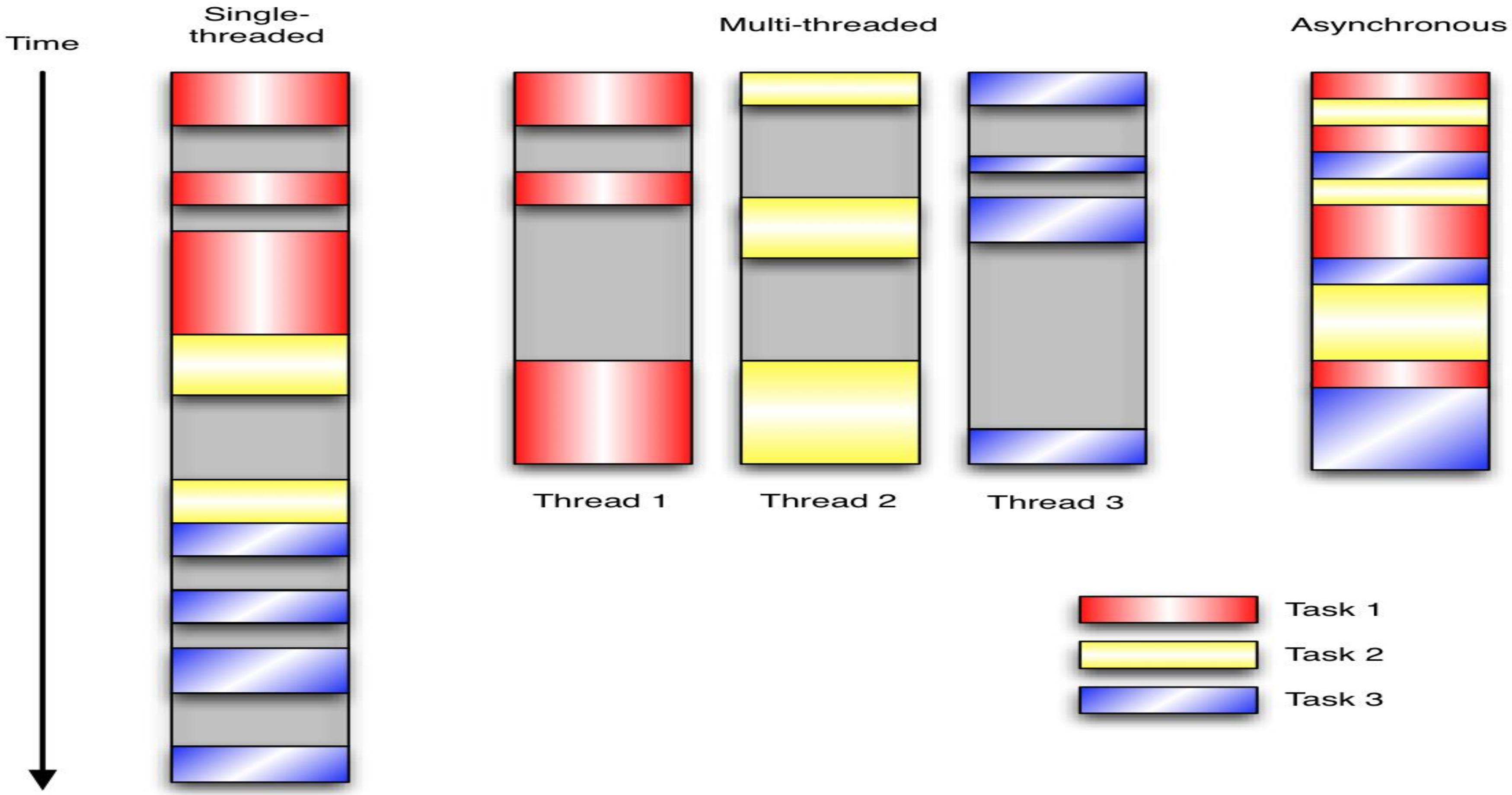
Twisted

- In 2000, glyph, the creator of Twisted, was working on a text-based multiplayer game called Twisted Reality.
- It was a big mess of threads, 3 per connection, in Java.
- There was a thread for input that would block on reads, a thread for output that would block on some kind of write, and a "logic" thread that would sleep while waiting for timers to expire or events to queue.
- As players moved through the virtual landscape and interacted, threads were deadlocking, caches were getting corrupted, and the locking logic was never quite right—the use of threads had made the software complicated, buggy, and hard to scale.

Twisted

- Over time, Twisted Reality the game became Twisted the networking platform, which would do what existing networking platforms in Python didn't:
- Use event-driven programming instead of multi-threaded programming.
- Be cross-platform: provide a uniform interface to the event notification systems exposed by major operating systems.
- Be "batteries-included": provide implementations of popular application-layer protocols out of the box, so that Twisted is immediately useful to developers.
- Conform to RFCs, and prove conformance with a robust test suite.
- Make it easy to use multiple networking protocols together.
- Be extensible.

Twisted


Time
Single-threaded
Multi-threaded
Asynchronous
Thread 1
Thread 2
Thread 3
Task 1
Task 2
Task 3

Twisted Architecture

- Twisted is an event-driven networking engine. Event-driven programming is so integral to Twisted's design philosophy that it is worth taking a moment to review what exactly event-driven programming means.

- Event-driven programming is a programming paradigm in which program flow is determined by external events. It is characterized by an event loop and the use of callbacks to trigger actions when events happen. Two other common programming paradigms are (single-threaded) synchronous and multi-threaded programming.

- Let's compare and contrast single-threaded, multi-threaded, and event-driven programming models with an example. Figure shows the work done by a program over time under these three models. The program has three tasks to complete, each of which blocks while waiting for I/O to finish. Time spent blocking on I/O is greyed out.

Twisted Architecture

- In the single-threaded synchronous version of the program, tasks are performed serially.
- If one task blocks for a while on I/O, all of the other tasks have to wait until it finishes and they are executed in turn.
- This definite order and serial processing are easy to reason about, but the program is unnecessarily slow if the tasks don't depend on each other, yet still have to wait for each other.
- In the threaded version of the program, the three tasks that block while doing work are performed in separate threads of control.
- These threads are managed by the operating system and may run concurrently on multiple processors or interleaved on a single processor.
- This allows progress to be made by some threads while others are blocking on resources. This is often more time-efficient than the analogous synchronous program,

Twisted Architecture

- but one has to write code to protect shared resources that could be accessed concurrently from multiple threads.
- Multi-threaded programs can be harder to reason about because one now has to worry about thread safety via process serialization (locking), reentrancy, thread-local storage, or other mechanisms, which when implemented improperly can lead to subtle and painful bugs.

Twisted Architecture

- The event-driven version of the program interleaves the execution of the three tasks, but in a single thread of control.
- When performing I/O or other expensive operations, a callback is registered with an event loop, and then execution continues while the I/O completes.
- The callback describes how to handle an event once it has completed.
- The event loop polls for events and dispatches them as they arrive, to the callbacks that are waiting for them.
- This allows the program to make progress when it can without the use of additional threads.
- Event-driven programs can be easier to reason about than multi-threaded programs because the programmer doesn't have to worry about thread safety.

Twisted Architecture

- The event-driven model is often a good choice when there are:
- many tasks, that are…
- largely independent (so they don't have to communicate with or wait on each other), and…
- some of these tasks block while waiting on events.

Twisted Architecture

- Twisted implements the reactor design pattern, which describes demultiplexing and dispatching events from multiple sources to their handlers in a single-threaded environment.
- The core of Twisted is the reactor event loop.
- The reactor knows about network, file system, and timer events.
- It waits on and then handles these events, abstracting away platform-specific behavior and presenting interfaces to make responding to events anywhere in the network stack easy.

Reactor

- Reactor Basics
- The reactor is the core of the event loop within Twisted – the loop which drives applications using Twisted.
- The event loop is a programming construct that waits for and dispatches events or messages in a program.
- It works by calling some internal or external “event provider”, which generally blocks until an event has arrived, and then calls the relevant event handler (“dispatches the event”).
- The reactor provides basic interfaces to a number of services, including network communications, threading, and event dispatching.

Reactor

- For information about using the reactor and the Twisted event loop, see:
- the event dispatching how tos: Scheduling and Using Deferreds ;
- the communication how tos: TCP servers , TCP clients , UDP networking and Using processes ; and
- Using threads .

Reactor

- The reactor usually implements a set of interfaces, but depending on the chosen reactor and the platform, some of the interfaces may not be implemented:

- IReactorCore : Core (required) functionality.
- IReactorFDSet : Use FileDescriptor objects.
- IReactorProcess : Process management. Read the Using Processes document for more information.
- IReactorSSL : SSL networking support.
- IReactorTCP : TCP networking support. More information can be found in the Writing Servers and Writing Clients documents.
- IReactorThreads : Threading use and management. More information can be found within Threading In Twisted .
- IReactorTime : Scheduling interface. More information can be found within Scheduling Tasks .
- IReactorUDP : UDP networking support. More information can be found within UDP Networking .
- IReactorUNIX : UNIX socket support.
- IReactorSocket : Third-party socket support..

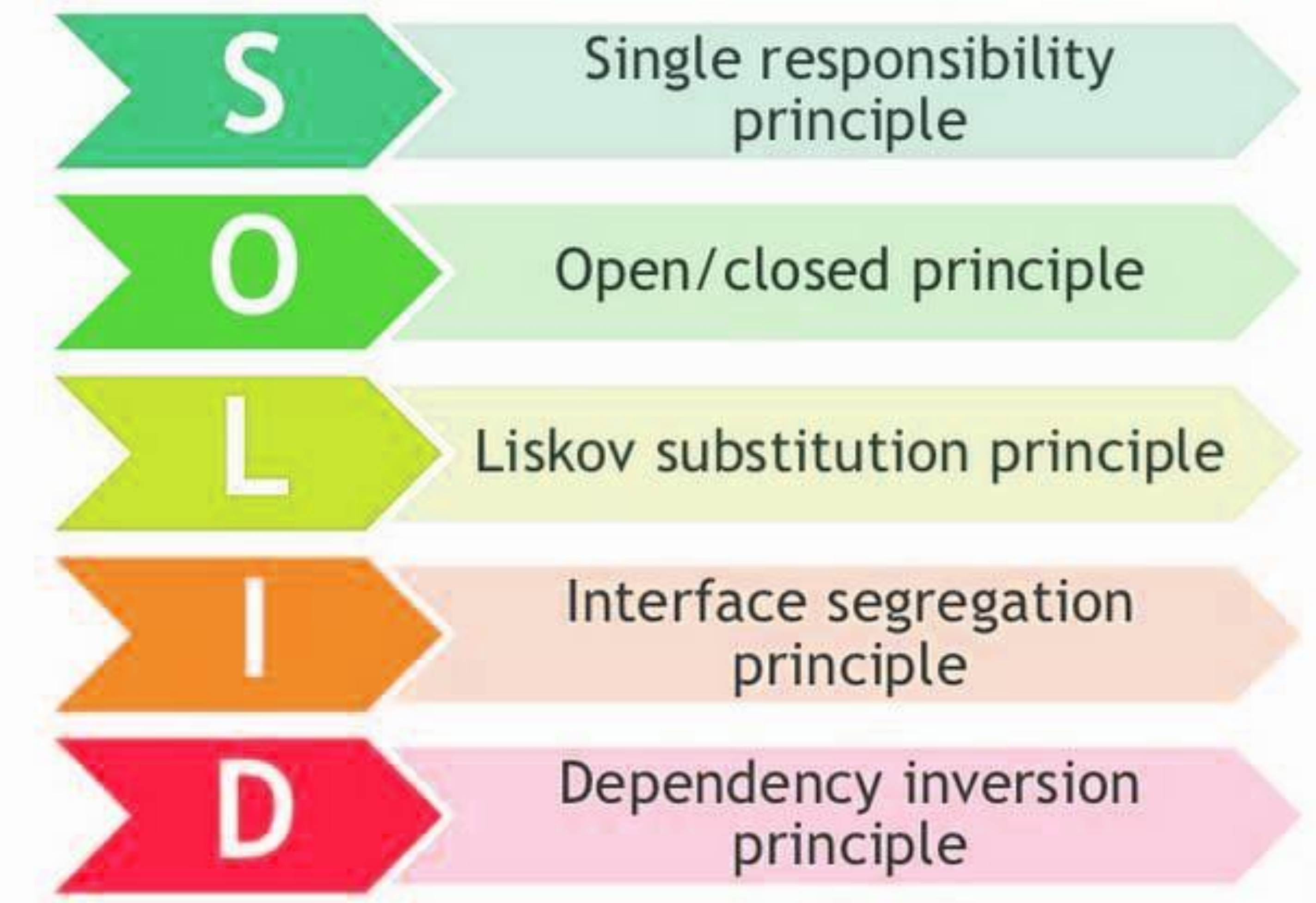
Scipy

- Python SciPy has modules for the following tasks:
- Optimization
- Linear algebra
- Integration
- Interpolation
- Special functions
- FFT
- Signal and Image processing
- ODE solvers

Scipy – sub packages

- cluster- Hierarchical clustering.
- constants- Physical constants and factors of conversion.
- fftpack- Algorithms for Discrete Fourier Transform.
- integrate- Routines for numerical integration.
- interpolate- Tools for interpolation.
- io- Input and Output tools for data.
- lib- Wrappers to external libraries.
- linalg- Routines for linear algebra.
- misc- Miscellaneous utilities like image reading and writing.
- ndimage- Functions for processing multidimensional images.
- optimize- Algorithms for optimization.
- signal- Tools for processing signal.
- sparse- Algorithms for sparse matrices.
- spatial- KD-trees, distance functions, nearest neighbors.
- special- Special functions.
- stats- Functions to perform statistics.
- weave- Functionality that lets you write C++/C code in the form of multiline strings.

SOLID Design Pattern



S — Single responsibility principle

O — Open/closed principle

L — Liskov substitution principle

I — Interface segregation principle

D — Dependency inversion principle

Single Responsibility Principle

- As the single responsibility principle states, a class should have only one reason to change.
- This principle says that when we develop classes, it should cater to the given functionality well.
- If a class is taking care of two functionalities, it is better to split them. It refers to functionality as a reason to change.

Single Responsibility Principle

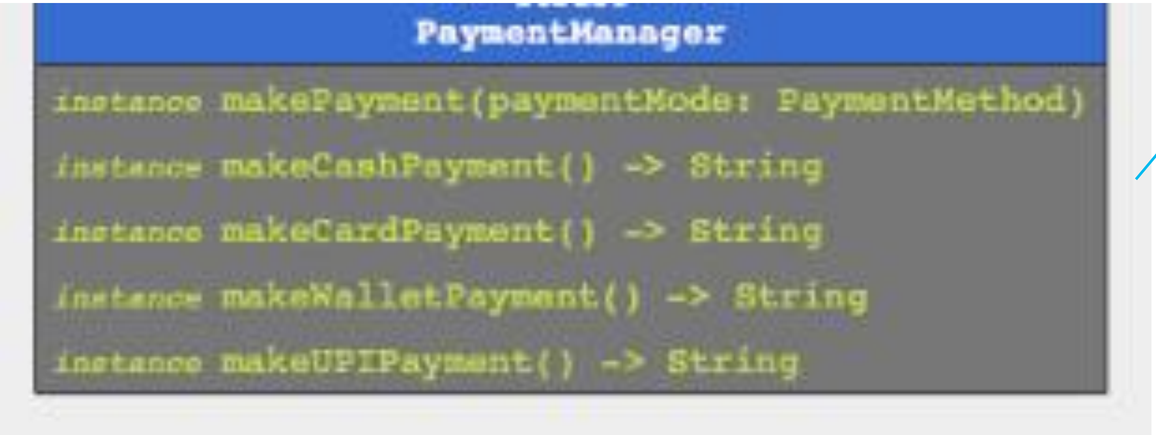
- Let's assume we need an object to keep an email message. We are going to use the IEmail. At the first sight everything looks just fine.
- IEmail and Email class have 2 responsibilities (reasons to change). One would be the use of the class in some email protocols such as pop3 or imap.
- If other protocols must be supported the objects should be serialized in another manner and code should be added to support new protocols.
- Another one would be for the Content field. Even if content is a string maybe we want in the future to support HTML or other formats.
- If we keep only one class, each change for a responsibility might affect the other one:
  - Adding a new protocol will create the need to add code for parsing and serializing the content for each type of field.
  - Adding a new content type (like html) make us to add code for each protocol implemented.

The open/close principle

- The open/close principle states that *classes or objects and methods should be open for extension but closed for modifications*.
- What this means in simple language is, when you develop your software application, make sure that you write your classes or modules in a generic way so that whenever you feel the need to extend the behavior of the class or object, then you shouldn't have to change the class itself.
- Rather, a simple extension of the class should help you build the new behavior.

The open/close principle

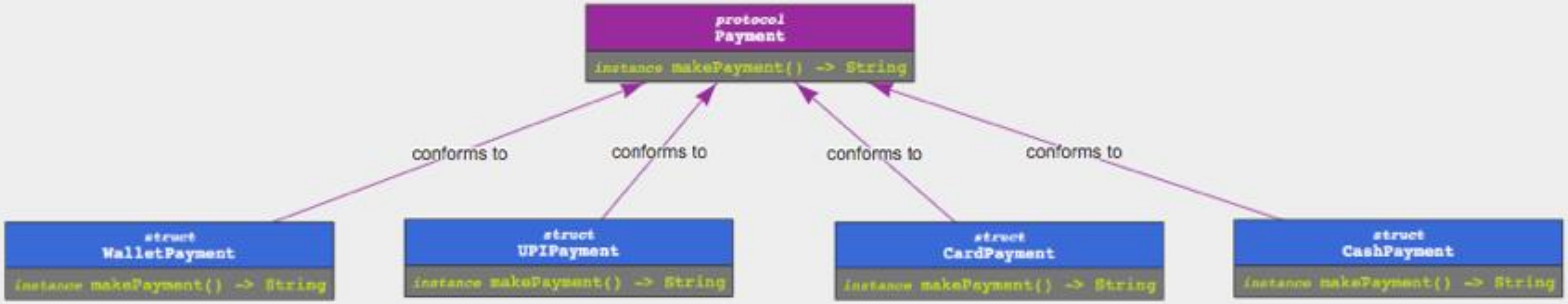
Not applying open/close principle



PaymentManager

```
instance makePayment(paymentMode: PaymentMethod)
instance makeCashPayment() -> String
instance makeCardPayment() -> String
instance makeWalletPayment() -> String
instance makeUPIPayment() -> String
```

Open close principle



protocol
Payment

instance makePayment() -> String

conforms to

conforms to

conforms to

conforms to

struct
WalletPayment

instance makePayment() -> String

struct
UPIPayment

instance makePayment() -> String

struct
CardPayment

instance makePayment() -> String

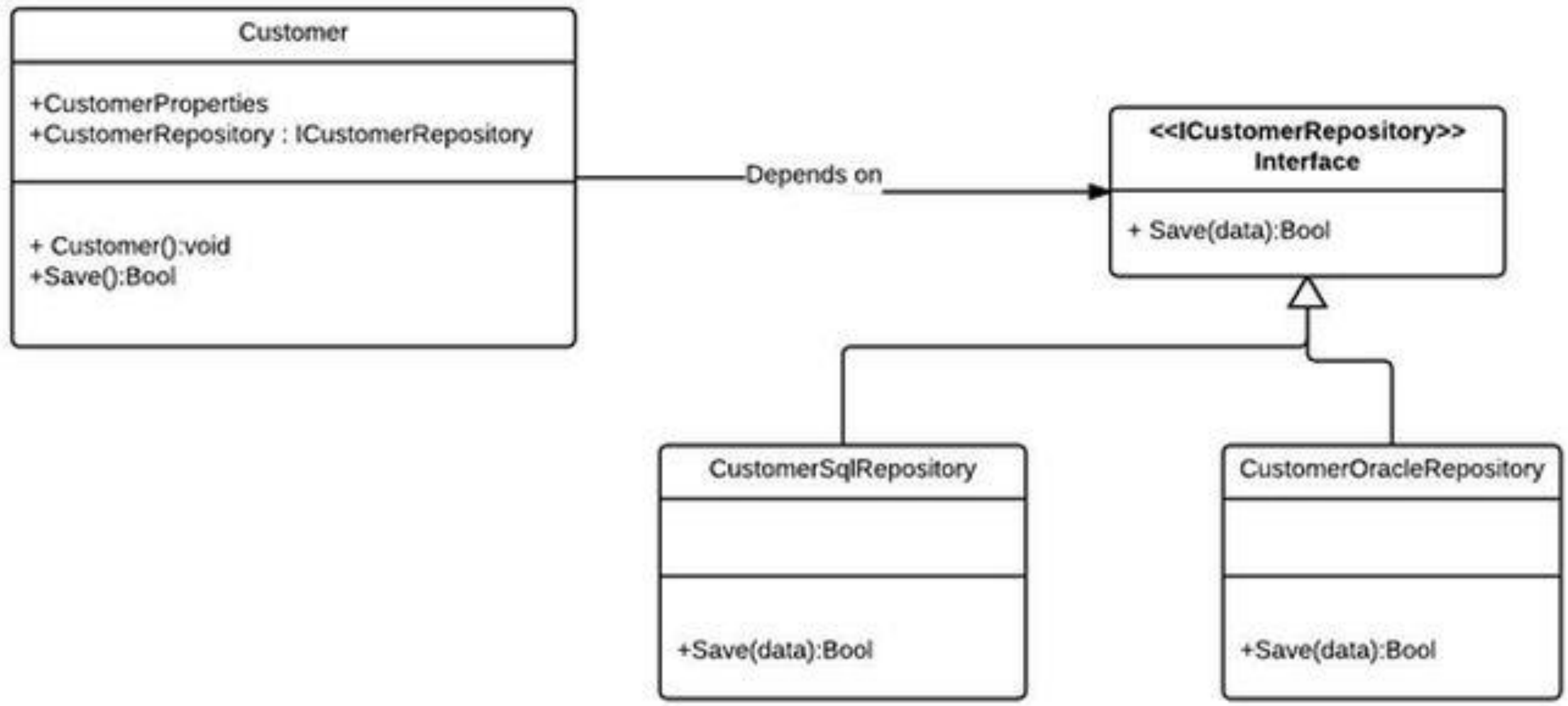
struct
CashPayment

instance makePayment() -> String

The inversion of control principle

- The inversion of control principle states that high-level modules shouldn't be dependent on low-level modules;
- They should both be dependent on abstractions. Details should depend on abstractions and not the other way round.

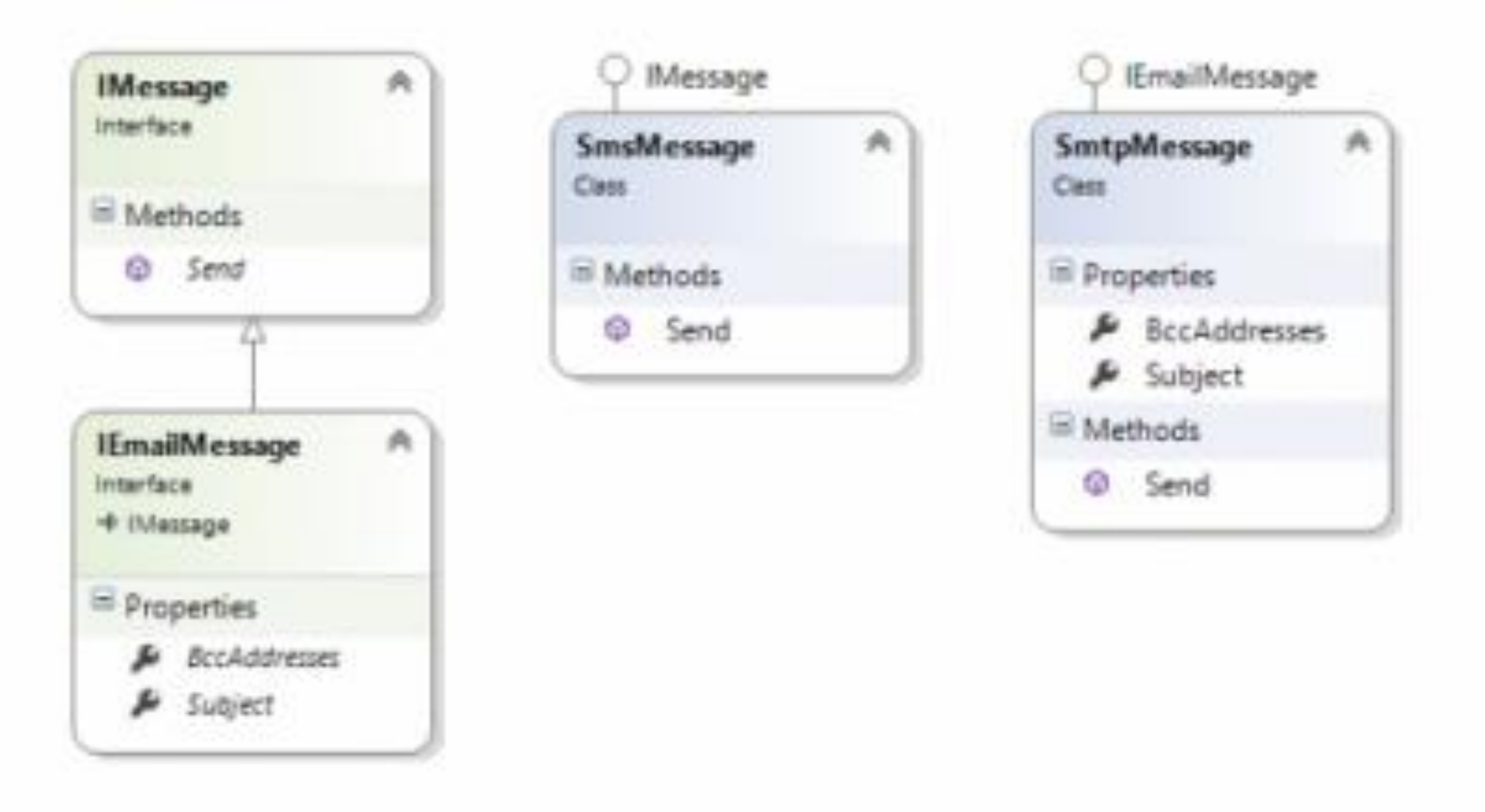
The inversion of control principle



The interface segregation principle

- As the interface segregation principle states, clients should not be forced to depend on interfaces they don't use.
- This principle talks about software developers writing their interfaces well.
- For instance, it reminds the developers/architects to develop methods that relate to the functionality.
- If there is any method that is not related to the interface, the class dependent on the interface has to implement it unnecessarily.

The interface segregation principle



Singleton

- Singleton provides you with a mechanism to have one, and only one, object of a given type and provides a global point of access.
- Hence, Singletons are typically used in cases such as logging or database operations, printer spoolers, and many others, where there is a need to have only one instance that is available across the application to avoid conflicting requests on the same resource.
- For example, we may want to use one database object to perform operations on the DB to maintain data consistency or one object of the logging class across multiple services to dump log messages in a particular log file sequentially.
- In brief, the intentions of the Singleton design pattern are as follows:
- Ensuring that one and only one object of the class gets created
- Providing an access point for an object that is global to the program
- Controlling concurrent access to resources that are shared

Design Pattern

- 1) Creational Pattern
- Factory Method Pattern
- Abstract Factory Pattern
- Singleton Pattern
- Prototype Pattern
- Builder Pattern
- Object Pool Pattern

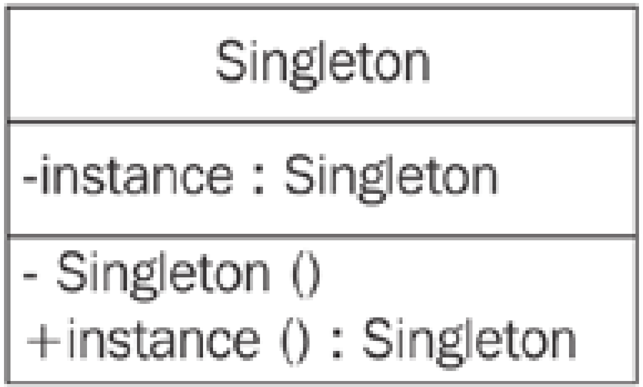
Design Pattern

- 2) Structural Pattern
- Adapter Pattern
- Bridge Pattern
- Composite Pattern
- Decorator Pattern
- Facade Pattern
- Flyweight Pattern
- proxy Pattern

Design Pattern

- 3) Behavioral Pattern
- Chain of Responsibility
- Command Pattern
- Interpreter Pattern
- Iterator Pattern
- Mediator Pattern
- Memento Pattern
- Observer Pattern
- State Pattern
- Strategy Pattern
- Template Pattern

Singleton



Singleton

-instance : Singleton

- Singleton ()

+instance () : Singleton

Singleton

- A simple way of implementing Singleton is by making the constructor private and creating a static method that does the object initialization.
- This way, one object gets created on the first call and the class returns the same object thereafter.
- In Python, we will implement it in a different way as there's no option to create private constructors.

Singleton Disadvantages

- Global variables can be changed by mistake at one place and, as the developer may think that they have remained unchanged, the variables get used elsewhere in the application.
- Multiple references may get created to the same object. As Singleton creates only one object, multiple references can get created at this point to the same object.
- All classes that are dependent on global variables get tightly coupled as a change to the global data by one class can inadvertently impact the sother class.

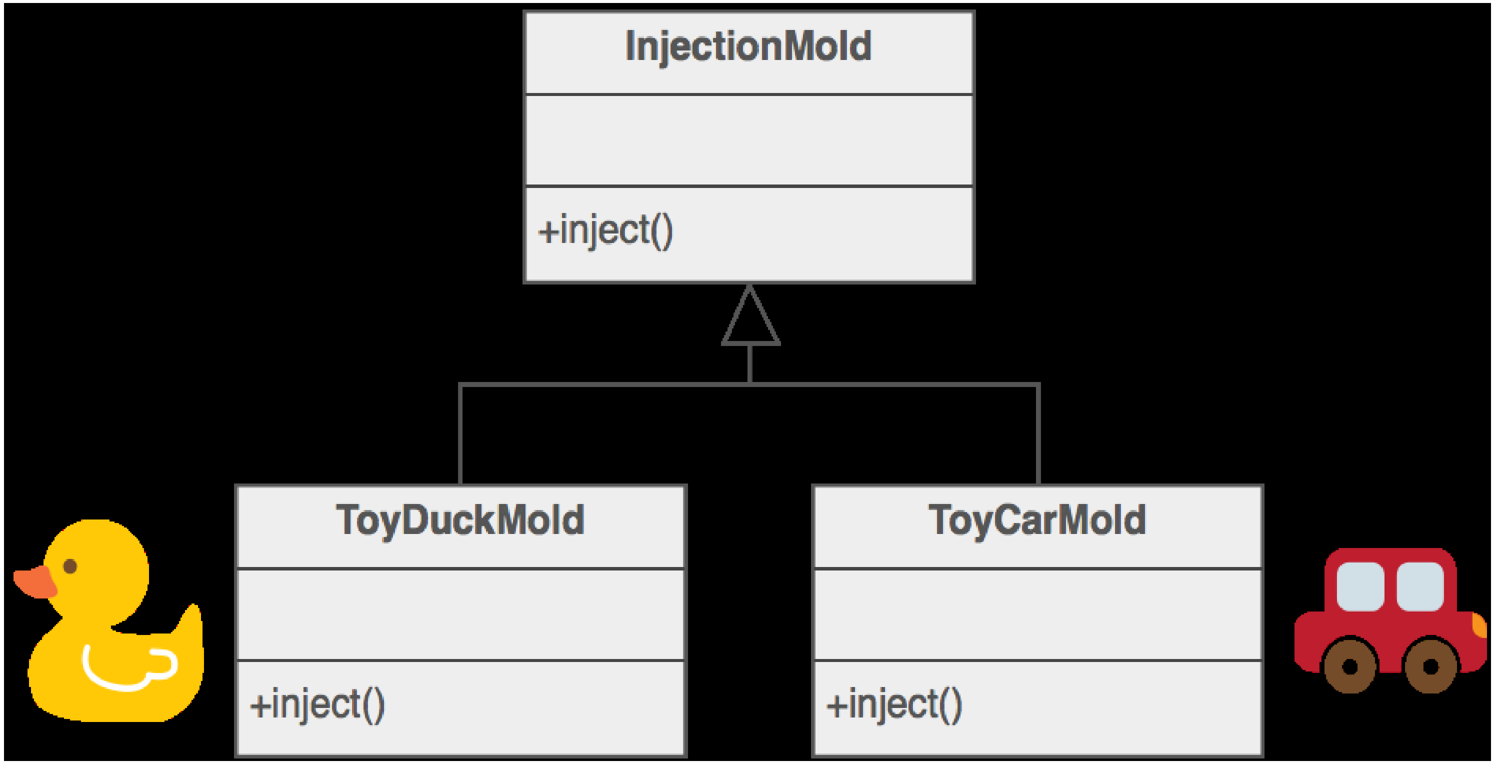
Factory Pattern

- The term factory means a class that is responsible for creating objects of other types.
- Typically, the class that acts as a factory has an object and methods associated with it.
- The client calls this method with certain parameters; objects of desired types are created in turn and returned to the client by the factory.

Factory Pattern

- A factory provides certain advantages that are listed here:
- The first advantage is loose coupling in which object creation can be independent of the class implementation.
- The client need not be aware of the class that creates the object which, in turn, is utilized by the client. It is only necessary to know the interface, methods, and parameters that need to be passed to create objects of the desired type. This simplifies implementations for the client.
- Adding another class to the factory to create objects of another type can be easily done without the client changing the code. At a minimum, the client needs to pass just another parameter.
- The factory can also reuse the existing objects. However, when the client does direct object creation, this always creates a new object.

Factory Pattern



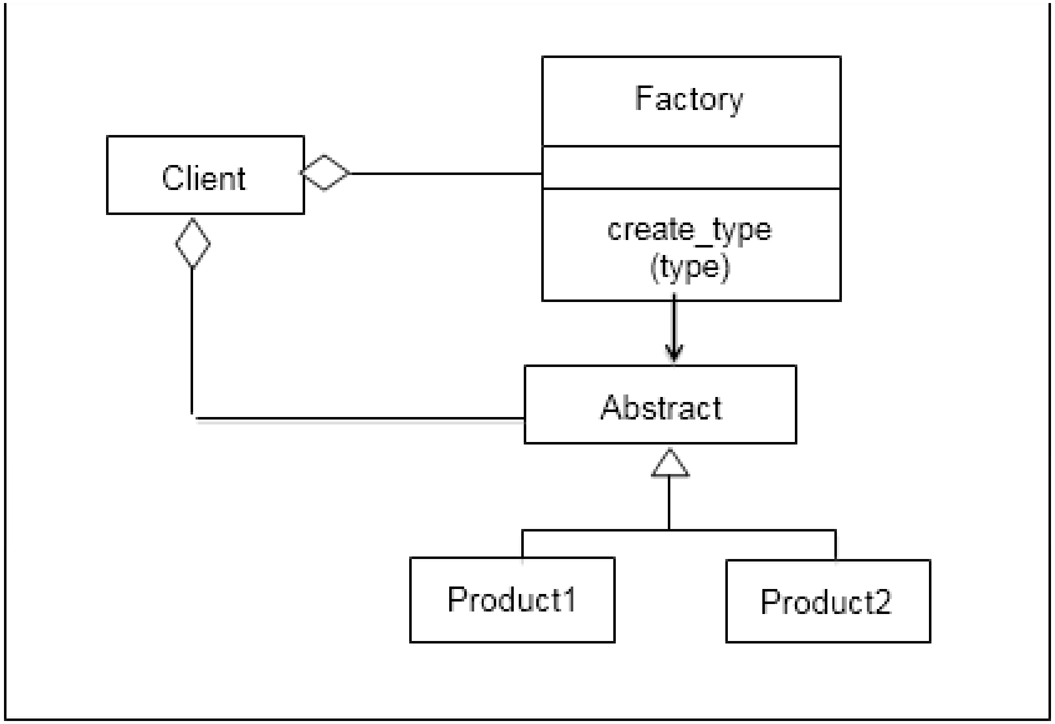
Factory Pattern

- **Simple Factory pattern**: This allows interfaces to create objects without exposing the object creation logic.
- **Factory method pattern**: This allows interfaces to create objects, but defers the decision to the subclasses to determine the class for object creation.
- **Abstract Factory pattern**: An Abstract Factory is an interface to create related objects without specifying/exposing their classes. The pattern provides objects of another factory, which internally creates other objects.

The Simple Factory pattern

- The Factory helps create objects of different types rather than direct object instantiation.
- The client class uses the Factory class, which has the create_type() method.
- When the client calls the create_type() method with the type parameters, based on the parameters passed, the Factory returns **Product1** or **Product2**:

Factory Pattern



Factory Method Pattern

- We define an interface to create objects, but instead of the factory being responsible for the object creation, the responsibility is deferred to the subclass that decides the class to be instantiated.
- The Factory method creation is through inheritance and not through instantiation.
- The Factory method makes the design more customizable. It can return the same instance or subclass rather than an object of a certain type (as in the simple factory method).

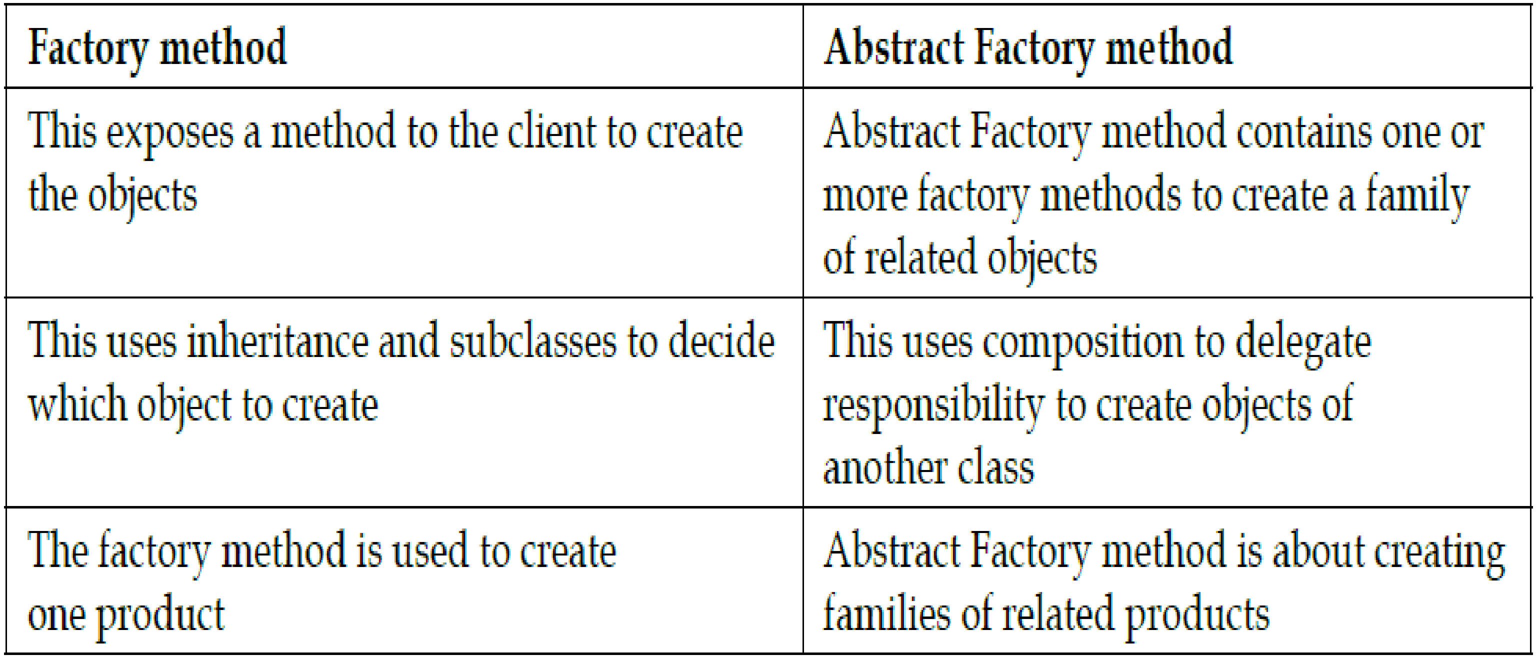
Advantages of the Factory method pattern

- It brings in a lot of flexibility and makes the code generic, not being tied to a certain class for instantiation.
- There's loose coupling, as the code that creates the object is separate from the code that uses it.
- The client need not bother about what argument to pass and which class to instantiate. The addition of new classes is easy and involves low maintenance.

The Abstract Factory pattern

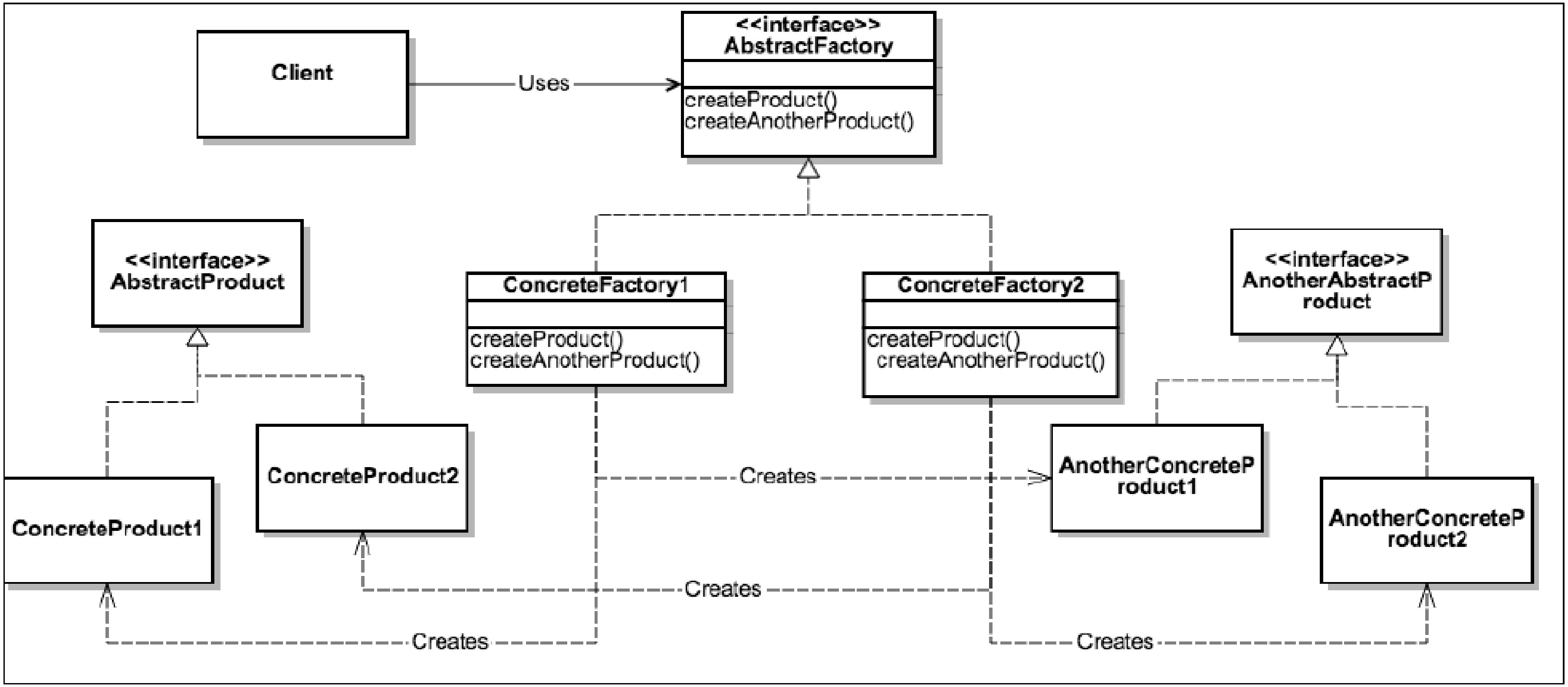
- The main objective of the Abstract Factory pattern is to provide an interface to create families of related objects without specifying the concrete class.
- While the factory method defers the creation of the instance to the subclasses, the goal of Abstract Factory method is to create families of related objects.

The Factory method versus Abstract Factory method



Factory method	Abstract Factory method
This exposes a method to the client to create the objects	Abstract Factory method contains one or more factory methods to create a family of related objects
This uses inheritance and subclasses to decide which object to create	This uses composition to delegate responsibility to create objects of another class
The factory method is used to create one product	Abstract Factory method is about creating families of related products

The Abstract Factory pattern



Prototype Pattern

- The **Prototype design pattern** helps us with creating object clones. In its simplest version, the Prototype pattern is just a clone() function that accepts an object as an input parameter and returns a clone of it.

- "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."
- In Python, this can be done using the copy.deepcopy() function.

Façade Design Pattern

- The façade is generally referred to as the face of the building, especially an attractive one.
- It can be also referred to as a behavior or appearance that gives a false idea of someone's true feelings or situation.
- When people walk past a façade, they can appreciate the exterior face but aren't aware of the complexities of the structure within. This is how a façade pattern is used.
- Façade hides the complexities of the internal system and provides an interface to the client that can access the system in a very simplified way.

Façade Design Pattern

- Consider the example of a storekeeper. Now, when you, as a customer, visit a store to buy certain items, you're not aware of the layout of the store.
- You typically approach the storekeeper, who is well aware of the store system. Based on your requirements, the storekeeper picks up items and hands them over to you.
- Isn't this easy? The customer need not know how the store looks and s/he gets the stuff done through a simple interface, the storekeeper.

Façade Design Pattern

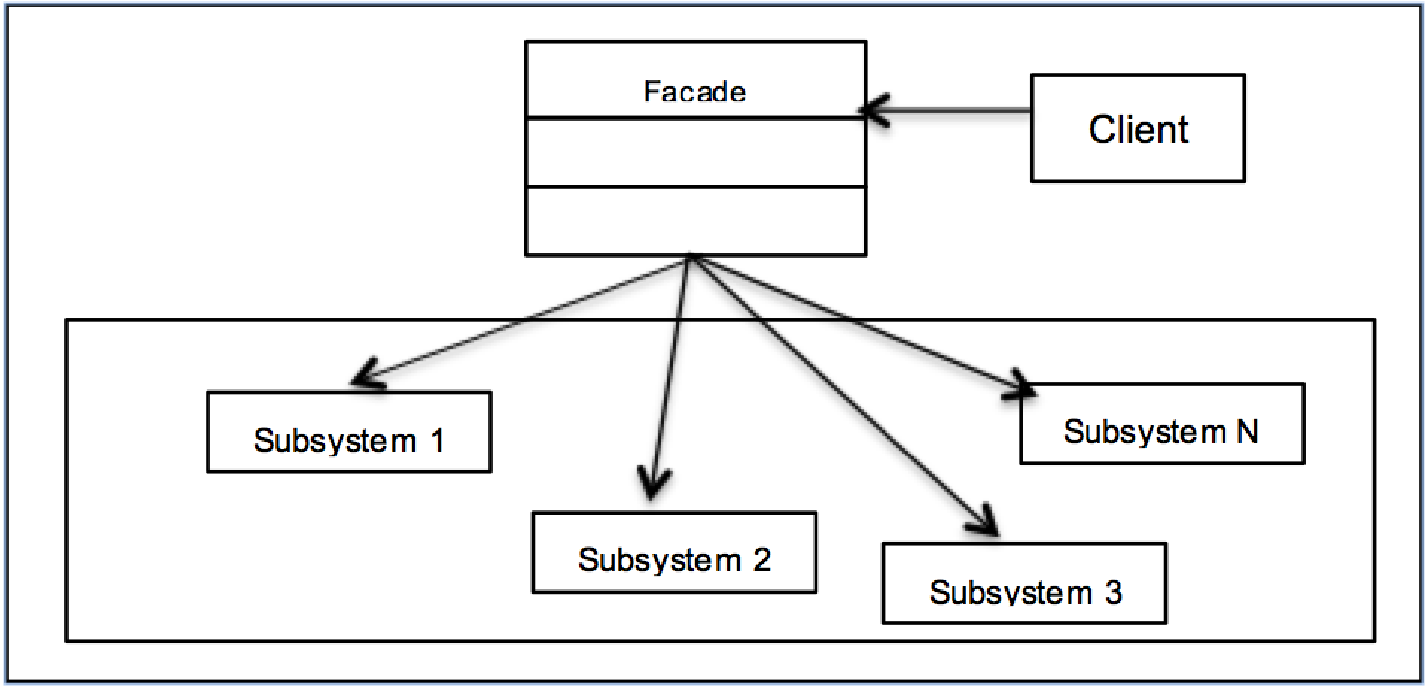
The Façade design pattern essentially does the following:

- It provides a unified interface to a set of interfaces in a subsystem and defines a high-level interface that helps the client use the subsystem in an easy way.

- Façade discusses representing a complex subsystem with a single interface object. It doesn't encapsulate the subsystem but actually combines the underlying subsystems.

- It promotes the decoupling of the implementation with multiple clients.

Façade Design Pattern



Facade

- It is an interface that knows which subsystems are responsible for a request
- It delegates the client's requests to the appropriate subsystem objects using composition
- For example, if the client is looking for some work to be accomplished, it need not have to go to individual subsystems but can simply contact the interface (Façade) that gets the work done.

Implementing the Façade pattern in the real world

- Consider that you have a marriage in your family and you are in charge of all the arrangements.
- That's a tough job on your hands. You have to book a hotel or place for marriage, talk to a caterer for food arrangements, organize a florist for all the decorations, and finally handle the musical arrangements expected for the event.

Implementing the Façade pattern in the real world

- In yesteryears, you'd have done all this by yourself, for example by talking to the relevant folks, coordinating with them, negotiating on the pricing, but now life is simpler.
- You go and talk to an event manager who handles this for you. S/he will make sure that they talk to the individual service providers and get the best deal for you.

Implementing the Façade pattern in the real world

- **Client**: It's you who need all the marriage preparations to be completed in time before the wedding. They should be top class and guests should love the celebrations.
- **Façade**: The event manager who's responsible for talking to all the folks that need to work on specific arrangements such as food, and flower decorations, among others
- **Subsystems**: They represent the systems that provide services such as catering, hotel management, and flower decorations

Proxy Pattern

- Proxy, in general terms, is a system that intermediates between the seeker and provider.
- Seeker is the one that makes the request, and provider delivers the resources in response to the request.
- In the web world, we can relate this to a proxy server. The clients (users in the World Wide Web), when they make a request to the website, first connect to a proxy server asking for resources such as a web page.
- The proxy server internally evaluates this request, sends it to an appropriate server, and gets back the response, which is then delivered to the client.
- Thus, a proxy server encapsulates requests, enables privacy, and works well in distributed architectures.

Proxy Pattern

- In the context of design patterns, Proxy is a class that acts as an interface to real objects.
- Objects can be of several types such as network connections, large objects in memory and file, among others.
- In short, Proxy is a wrapper or agent object that wraps the real serving object.
- Proxy could provide additional functionality to the object that it wraps and doesn't change the object's code.
- The main intention of the Proxy pattern is to provide a surrogate or placeholder for another object in order to control access to a real object.

Proxy Pattern Scenarios

- It represents a complex system in a simpler way. For example, a system that involves multiple complex calculations or procedures should have a simpler interface that can act as a proxy for the benefit of the client.
- It adds security to the existing real objects. In many cases, the client is not allowed to access the real object directly. This is because the real object can get compromised with malicious activities. This way proxies act as a shield against malicious intentions and protect the real object.

Proxy Pattern Scenarios

- It provides a local interface for remote objects on different servers.
- A clear example of this is with the distributed systems where the client wants to run certain commands on the remote system, but the client may not have direct permissions to make this happen. So it contacts a local object (proxy) with the request, which is then executed by the proxy on the remote machine.

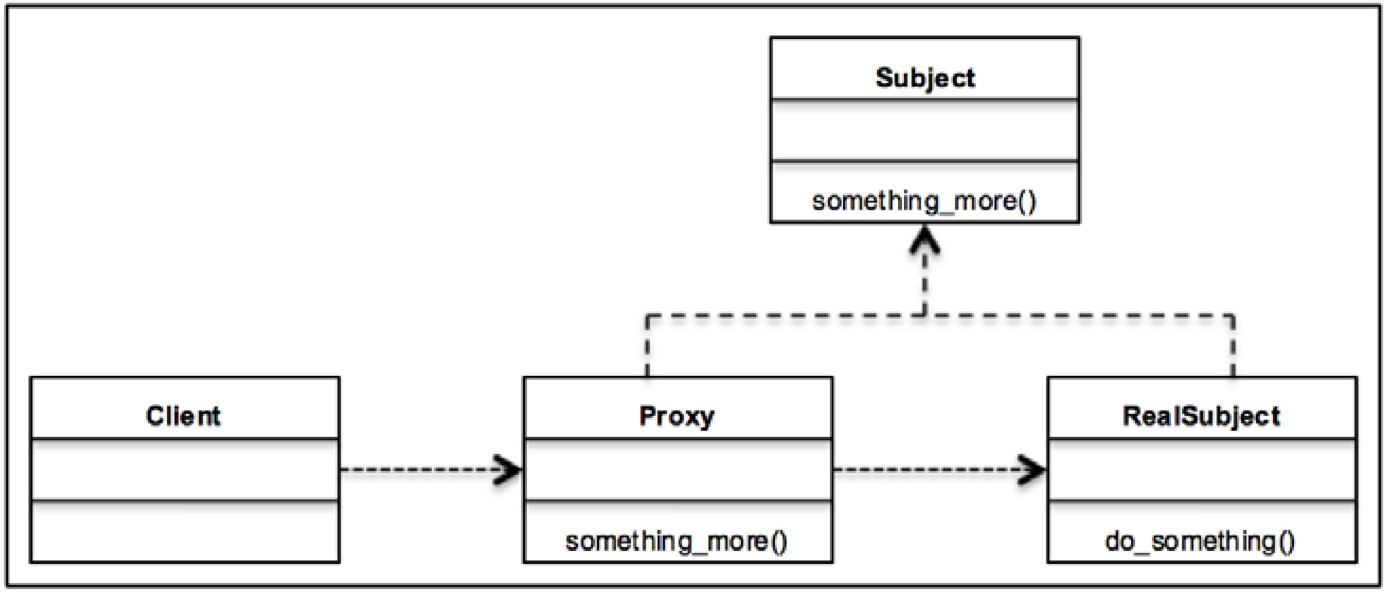
Proxy Pattern Scenarios

- It provides a light handle for a higher memory-consuming object. Sometimes, you may not want to load the main objects unless they're really necessary.
- This is because real objects are really heavy and may need high resource utilization.
- A classic example is that of profile pictures of users on a website. You're much better off showing smaller profile images in the list view, but of course, you'll need to load the actual image to show the detailed view of the user profile.
- .

Proxy Pattern Application

- Consider the example of an Actor and his Agent.
- When production houses want to approach an Actor for a movie, typically, they talk to the Agent and not to the Actor directly.
- Based on the schedule of the Actor and other engagements, the Agent gets back to the production house on the availability and interest in working in the movie.
- Now, in this scenario, instead of production houses directly talking to the Actor, the Agent acts as a Proxy that handles all the scheduling & payments for the Actor.

Proxy Pattern



Proxy Pattern

- Proxy: This maintains a reference that lets the Proxy access the real object. It provides an interface identical to the Subject so that Proxy can substitute the real subject. Proxies are also responsible for creating and deleting the RealSubject.
- Subject: It provides a representation for both, the RealSubject and Proxy. As Proxy and RealSubject implement Subject, Proxy can be used wherever RealSubject is expected.
- RealSubject: It defines the real object that the Proxy represents.

Proxy Pattern

- Proxy: It is a class that controls access to the RealSubject class. It handles the client's requests and is responsible for creating or deleting RealSubject.
- Subject/RealSubject: Subject is an interface that defines what RealSubject and Proxy should look like. RealSubject is an actual implementation of the Subject interface. It provides the real functionality that is then used by the client.
- Client: It accesses the Proxy class for the work to be accomplished. The Proxy class internally controls access to RealSubject and directs the work requested by Client.

Understanding different types of Proxies

- we can categorize them as virtual proxy, remote proxy, protective proxy, and smart proxy.
- **A virtual proxy**
- Here, you'll learn in detail about the virtual proxy. It is a placeholder for objects that are very heavy to instantiate.
- For example, you want to load a large image on your website. Now this request will take a long time to load.
- Typically, developers will create a placeholder icon on the web page suggesting that there's an image. However, the image will only be loaded when the user actually clicks on the icon thus saving the cost of loading a heavy image in memory.
- Thus, in virtual proxies, the real object is created when the client first requests or accesses the object.

Understanding different types of Proxies

- **A remote proxy**
- A remote proxy can be defined in the following terms. It provides a local representation of a real object that resides on a remote server or different address space.
- For example, you want to build a monitoring system for your application that has multiple web servers, DB servers, celery task servers, caching servers, among others.
- If we want to monitor the CPU and disk utilization of these servers, we need to have an object that is available in the context of where the monitoring application runs but can perform remote commands to get the actual parameter values. In such cases, having a remote proxy object that is a local representation of the remote object would help.

Understanding different types of Proxies

- **A protective proxy**
- This proxy controls access to the sensitive matter object of RealSubject.
- For example, in today's world of distributed systems, web applications have multiple services that work together to provide functionality.
- Now, in such systems, an authentication service acts as a protective proxy server that is responsible for authentication and authorization. I
- In this case, Proxy internally helps in protecting the core functionality of the website for unrecognized or unauthorized agents. Thus, the surrogate object checks that the caller has access permissions required to forward the request.

Understanding different types of Proxies

- **A smart proxy**
- Smart proxies interpose additional actions when an object is accessed. For example, consider that there's a core component in the system that stores states in a centralized location.
- Typically, such a component gets called by multiple different services to complete their tasks and can result in issues with shared resources. Instead of services directly invoking the core component, a smart proxy is built-in and checks whether the real object is locked before it is accessed in order to ensure that no other object can change it.

Proxy Scenario

- Let's say that you go to shop at a mall and like a nice denim shirt there. You would like to purchase the shirt but you don't have enough cash to do so.
- In yesteryears, you'd go to an ATM, take out the money, then come to the mall, and pay for it. Even earlier, you had a bank check for which you had to go to the bank, withdraw money, and then come back to pay for your expense.
- Thanks to the banks, we now have something called a debit card. So now, when you want to purchase something, you present your debit card to the merchant. When you punch in your card details, the money is debited in the merchant's account for your expense.

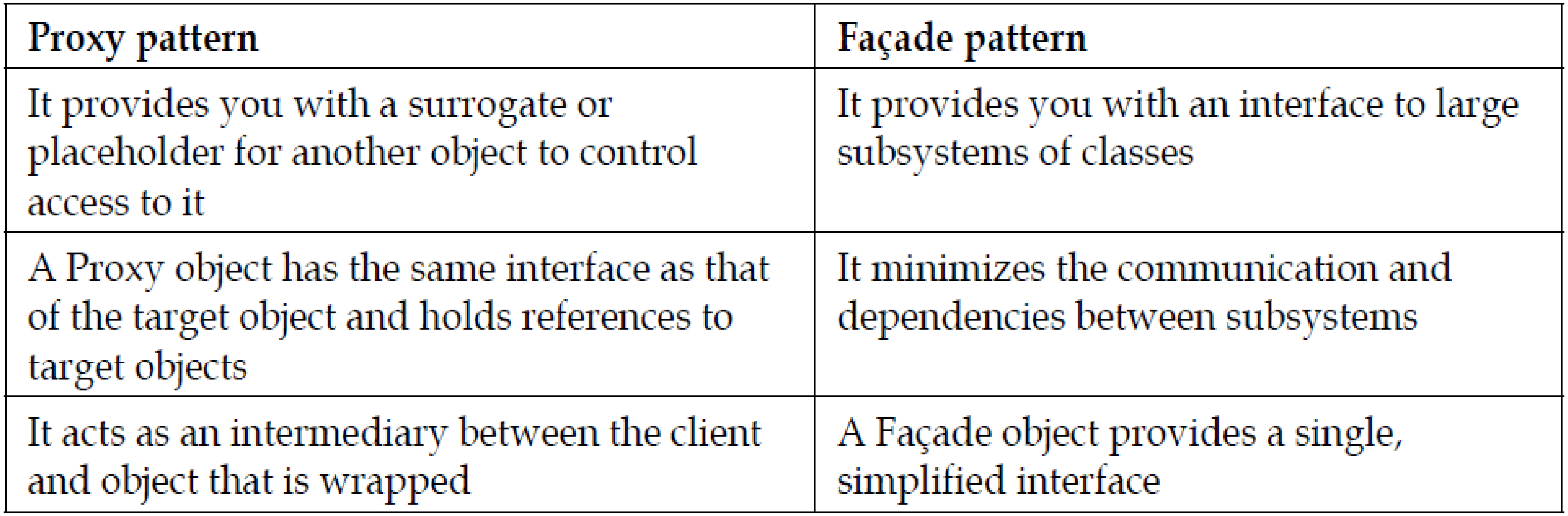
Proxy Scenario

- Your behavior is represented by the You class—the client
- To buy the shirt, the make_payment() method is provided by the class
- The special __init__() method calls the Proxy and instantiates it
- The make_payment() method invokes the Proxy's method internally to make the payment
- The __del__() method returns in case the payment is successful

Advantages

- Proxies can help improve the performance of the application by caching heavy objects or, typically, the frequently accessed objects.
- Proxies also authorize the access to RealSubject; thus, this pattern helps in delegation only if the permissions are right
- Remote proxies also facilitate interaction with remote servers that can work as network connections and database connections and can be used to monitor systems

Comparing the Façade and Proxy patterns



Proxy pattern	Façade pattern
It provides you with a surrogate or placeholder for another object to control access to it	It provides you with an interface to large subsystems of classes
A Proxy object has the same interface as that of the target object and holds references to target objects	It minimizes the communication and dependencies between subsystems
It acts as an intermediary between the client and object that is wrapped	A Façade object provides a single, simplified interface

Observer Pattern

- In the Observer design pattern, an object (Subject) maintains a list of dependents (Observers) so that the Subject can notify all the Observers about the changes that it undergoes using any of the methods defined by the Observer.
- In the world of distributed applications, multiple services interact with each other to perform a larger operation that a user wants to achieve.
- Services can perform multiple operations, but the operation they perform is directly or heavily dependent on the state of the objects of the service that it interacts with.

Observer Pattern

- Consider a use case for user registration where the user service is responsible for user operations on the website.
- Let's say that we have another service called e-mail service that observes the state of the user and sends e-mails to the user.
- For example, if the user has just signed up, the user service will call a method of the e-mail service that will send an e-mail to the user for account verification.
- If the account is verified but has fewer credits, the e-mail service will monitor the user service and send an e-mail alert for low credits to the user.

Observer Pattern

- Thus, if there's a core service in the application on which many other services are dependent, the core service becomes the Subject that has to be observed/monitored by the Observer for changes.
- The Observer should, in turn, make changes to the state of its own objects or take certain actions based on the changes that happen in the Subject.
- The above scenario, where the dependent service monitor's state changes in the core service, presents a classical case for the Observer design pattern.

Observer Pattern

- Consider the example of a blog. Let's suppose that you're a tech enthusiast who loves to read about the latest articles on Python on this blog.
- You subscribe to the blog. Like you, there would be multiple subscribers that are also registered with the blog. So, whenever there is a new blog, you get notified, or if there is a change on the published blog, you are also made aware of the edits.
- The way in which you're notified of the change can be an e-mail. Now if you apply this scenario to the Observer pattern, the blog is the Subject that maintains the list of subscribers or Observers.
- So when a new entry is added to the blog, all Observers are notified via e-mail or any other notification mechanism as defined by the Observer.

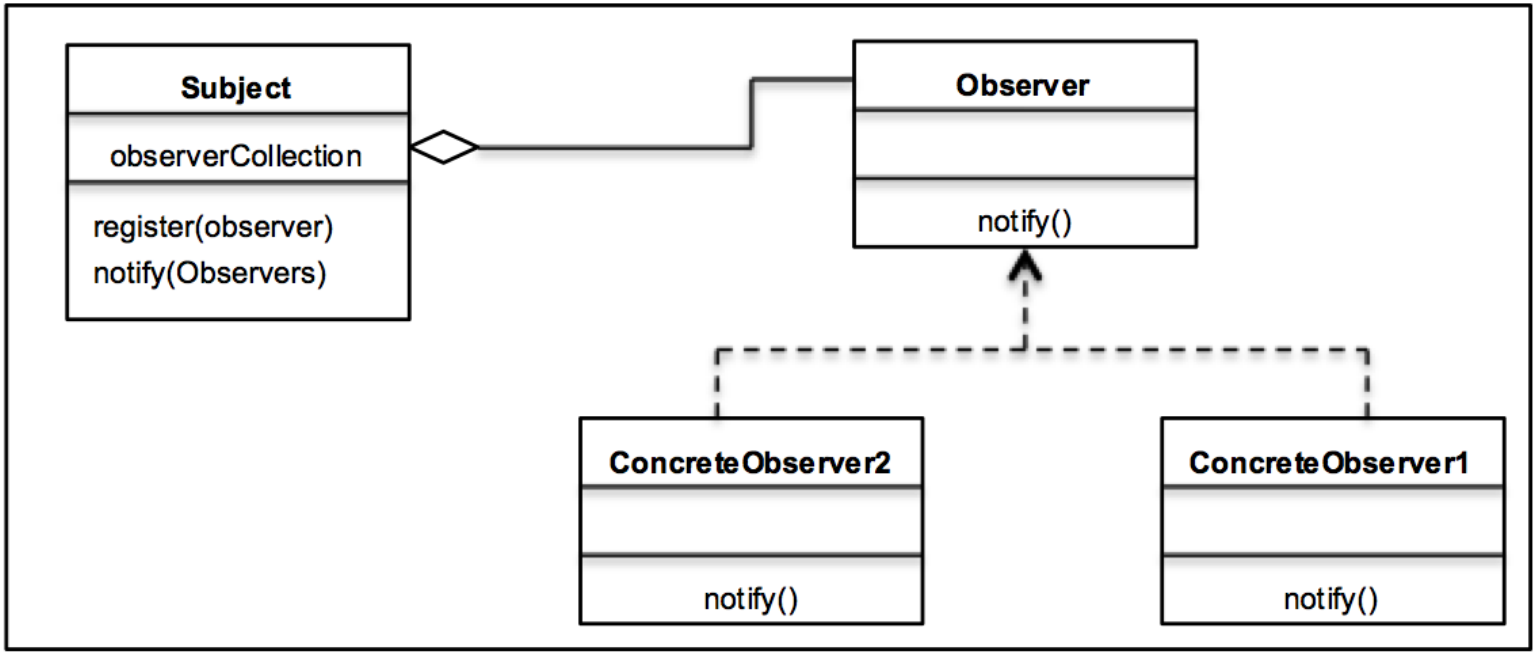
Observer Pattern

- The main intentions of the Observer pattern are as follows:
- It defines a one-to-many dependency between objects so that any change in one object will be notified to the other dependent objects automatically
- It encapsulates the core component of the Subject

Observer Pattern

- The Observer pattern is used in the following multiple scenarios:
- Implementation of the Event service in distributed systems
- A framework for a news agency
- The stock market also represents a great case for the Observer pattern

Observer Pattern



Observer Pattern

- News agencies typically gather news from various locations and publish them to the subscribers. Let's look at the design considerations for this use case.
- With information being sent/received in real time, a news agency should be able to publish the news as soon as possible to its subscribers.
- Additionally, because of the advancements in the technology industry, it's not just the newspapers, but also the subscribers that can be of different types such as an e-mail, mobile, SMS, or voice call. We should also be able to add any other type of subscriber in the future and budgeting for any new technology.

Observer Pattern

- **The Observer pattern methods**
- **The pull model**
- In the pull model, Observers play an active role as follows:
- The Subject broadcasts to all the registered Observers when there is any change
- The Observer is responsible for getting the changes or pulling data from the subscriber when there is an amendment
- The pull model is ineffective as it involves two steps—the first step where the Subject notifies the Observer and the second step where the Observer pulls the required data from the Subject

Observer Pattern

- **The push model**
- In the push model, the Subject is the one that plays a dominant role as follows:
- Unlike the pull model, the changes are pushed by the Subject to the Observer.
- In this model, the Subject can send detailed information to the Observer (even though it may not be needed). This can result in sluggish response times when a large amount of data is sent by the Subject but is never actually used by the Observer.
- Only the required data is sent from the Subject so that the performance is better.

The Observer pattern – advantages and disadvantages

- The Observer pattern provides you with the following advantages:
- It supports the principle of loose coupling between objects that interact with each other
- It allows sending data to other objects effectively without any change in the Subject or Observer classes
- Observers can be added/removed at any point in time

The Observer pattern – advantages and disadvantages

- The Observer interface has to be implemented by ConcreteObserver, which involves inheritance. There is no option for composition, as the Observer interface can be instantiated.
- If not correctly implemented, the Observer can add complexity and lead to inadvertent performance issues.
- In software application, notifications can, at times, be undependable and result in race conditions or inconsistency.

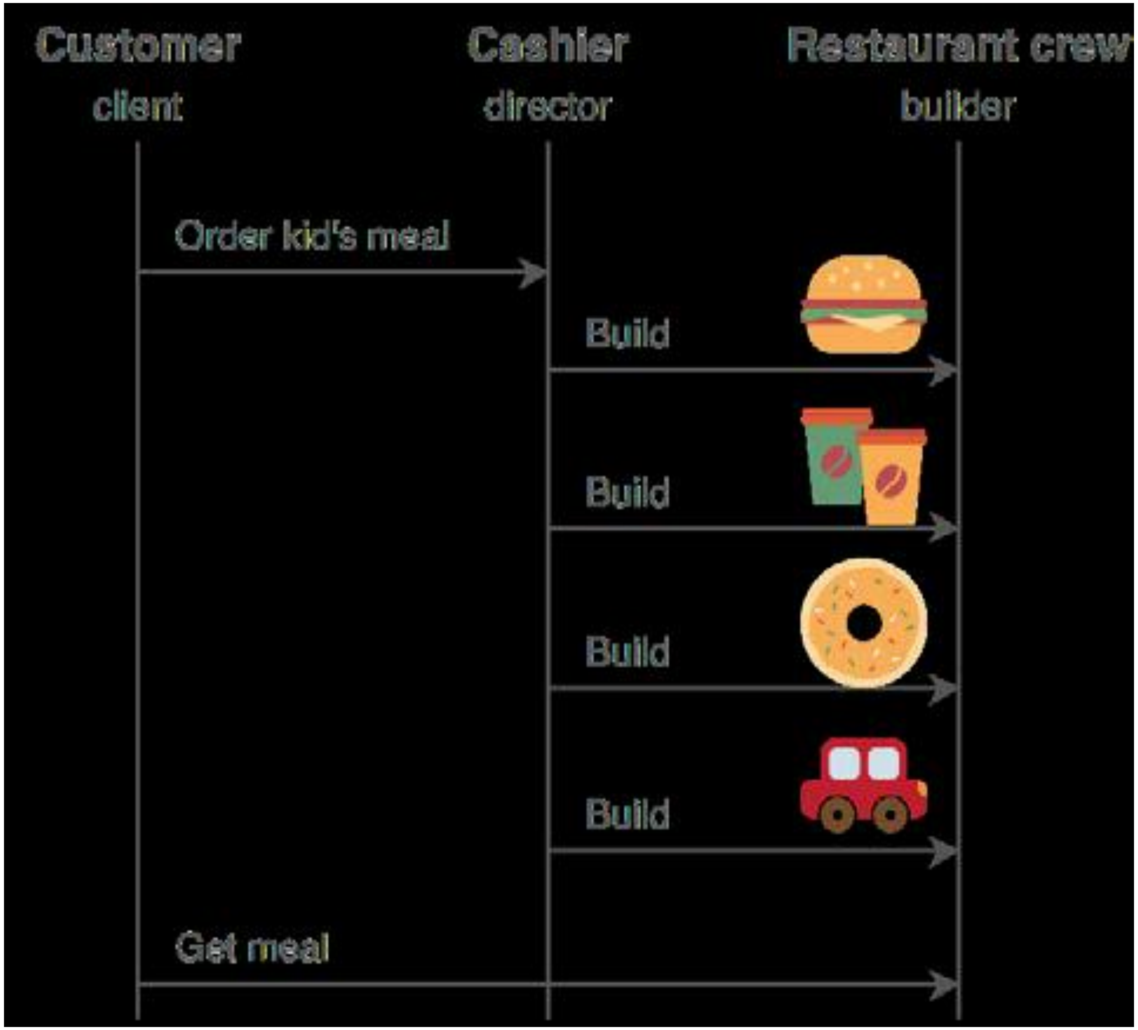
Builder Pattern

- The builder is responsible for creating the various parts of the complex object.
- In the HTML example, these parts are the title, heading, body, and the footer of the page.

Builder Pattern

- The Builder design pattern is used in fast-food restaurants. The same procedure is always used to prepare a burger and the packaging (box and paper bag), even if there are many different kinds of burgers (classic, cheeseburger, and more) and different packages (small-sized box, medium-sized box, and so forth).
- The difference between a classic burger and a cheeseburger is in the representation, and not in the construction procedure.
- The director is the cashier who gives instructions about what needs to be prepared to the crew, and the builder is the person from the crew that takes care of the specific order.

Builder Pattern



Adapter Pattern

- **Adapter** is a structural design pattern that helps us make two incompatible interfaces compatible.
- In such cases, we can write an extra layer that makes all the required modifications for enabling the communication between the two interfaces. This layer is called the Adapter.
- E-commerce systems are known examples. Assume that we use an e-commerce system that contains a calculate_total(order) function. The function calculates the total amount of an order, but only in **Danish Kroner** (**DKK**).

Adapter Pattern

- It is reasonable for our customers to ask us to add support for more popular currencies, such as **United States Dollars** (**USD**) and **Euros** (**EUR**).
- If we own the source code of the system we can extend it by adding new functions for doing the conversions from DKK to USD and from DKK to EUR.
- But what if we don't have access to the source code of the application because it is provided to us only as an external library? In this case, we can still use the library (for example, call its methods), but we cannot modify/extend it.
- The solution is to write a wrapper (also known as Adapter) that converts the data from the given DKK format to the expected USD or EUR format.

Adapter Pattern A real-life example

- **Grok** is a Python framework that runs on top of Zope 3 and focuses on agile development.
- The Grok framework uses Adapters for making it possible for existing objects to conform to specific APIs without the need to modify them [j.mp/grokada].
- The Python **Traits** package also uses the Adapter pattern for transforming an object that does not implement of a specific interface (or set of interfaces) to an object that does [j.mp/pytraitsad].

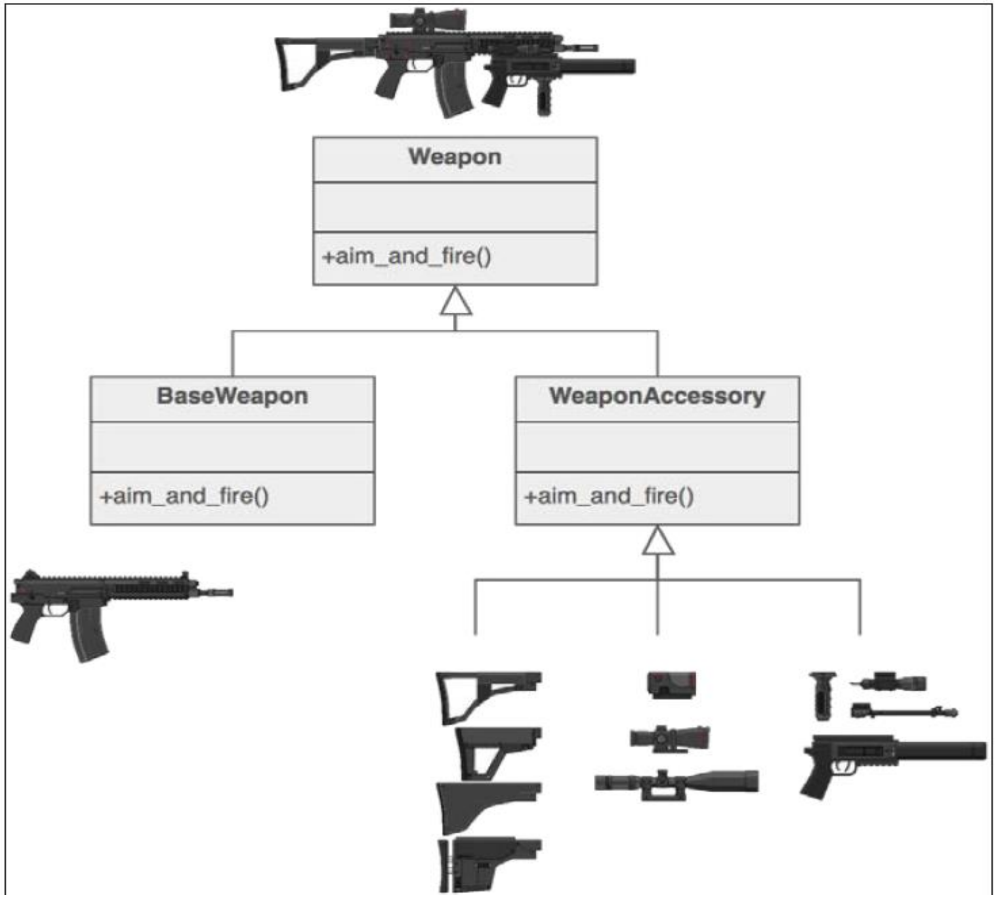
The Decorator Pattern

- A **Decorator** pattern can add responsibilities to an object dynamically, and in a transparent manner.
- This defines additional responsibilities for an object at runtime or dynamically. We add certain attributes to objects with an interface.
- Real examples of such extensions are: adding a silencer to a gun, using different camera lenses (in cameras with removable lenses), and so on.
- The Decorator pattern shines when used for implementing **cross-cutting concerns**

The Decorator Pattern

- Data validation
- Transaction processing (A transaction in this case is similar to a database transaction, in the sense that either all steps should be completed successfully, or the transaction should fail.)
- Caching
- Logging
- Monitoring
- Debugging
- Business rules
- Compression
- Encryption

The Decorator Pattern



The Decorator Pattern

- The Django framework uses decorators to a great extent. An example is the View decorator. Django's **View** decorators can be used for [j.mp/djangodec]:
- Restricting access to views based on the HTTP request
- Controlling the caching behavior on specific views
- Controlling compression on a per-view basis
- Controlling caching based on specific HTTP request headers

The Decorator Pattern

- The Grok framework also uses decorators for achieving different goals such as [j.mp/grokdeco]:
- Registering a function as an event subscriber
- Protecting a method with a specific permission
- Implementing the Adapter pattern

The Flyweight Pattern

- Flyweight design pattern is a technique used to minimize memory usage and improve performance by introducing data sharing between similar objects.
- A **Flyweight** is a shared object that contains state-independent, immutable (also known as intrinsic) data.
- The state-dependent, mutable (also known as extrinsic) data should not be part of Flyweight because this is information that cannot be shared since it differs per object.

The Flyweight Pattern

- We can think of Flyweight as caching in real life. For example, many bookstores have dedicated shelves with the newest and most popular publications.
- This is a cache. First, you can take a look at the dedicated shelves for the book you are looking for, and if you cannot find it, you can ask the librarian to assist you.

The Flyweight Pattern

- Memoization is an optimization technique that uses a cache to avoid recomputing results that were already computed in an earlier execution step.
- Memoization does not focus on a specific programming paradigm such as **object-oriented programming** (**OOP**).
- In Python, memoization can be applied on both methods and simple functions.
- Flyweight is an OOP-specific optimization design pattern that focuses on sharing object data.

The Model-View-Controller Pattern

- One of the design principles related to software engineering is the **Separation of Concerns** (**SoC**) principle.
- The idea behind the SoC principle is to split an application into distinct sections, where each section addresses a separate concern.
- The **Model-View-Controller** (**MVC**) pattern is nothing more than the SoC principle applied to OOP.
- The name of the pattern comes from the three main components used to split a software application: the model, the view, and the controller.
- MVC is considered an architectural pattern rather than a design pattern

The Model-View-Controller Pattern

- The difference between an architectural and a design pattern is that the former has a broader scope than the latter.
- A typical use of an application that uses MVC after the initial screen is rendered to the user is as follows:
- The user triggers a view by clicking (typing, touching, and so on) a button
- The view informs the controller about the user's action
- The controller processes user input and interacts with the model
- The model performs all the necessary validation and state changes, and informs the controller about what should be done
- The controller instructs the view to update and display the output appropriately, following the instructions given by the model

The Model-View-Controller Pattern

- The **web2py** web framework [j.mp/webtopy] is a lightweight Python framework that embraces the MVC pattern
- Django is also an MVC framework, although it uses different naming conventions. The controller is called view, and the view is called template. Django uses the name **Model-Template-View** (**MTV**).

The Model-View-Controller Pattern

- A model is considered smart because it:
- Contains all the validation/business rules/logic
- Handles the state of the application
- Has access to application data (database, cloud, and so on)
- Does not depend on the UI

The Model-View-Controller Pattern

- A controller is considered thin because it:
- Updates the model when the user interacts with the view
- Updates the view when the model changes
- Processes the data before delivering it to the model/view, if necessary

The Model-View-Controller Pattern

- A controller is considered thin because it:
- Updates the model when the user interacts with the view
- Updates the view when the model changes
- Processes the data before delivering it to the model/view, if necessary
- Does not display the data
- Does not access the application data directly
- Does not contain validation/business rules/logic

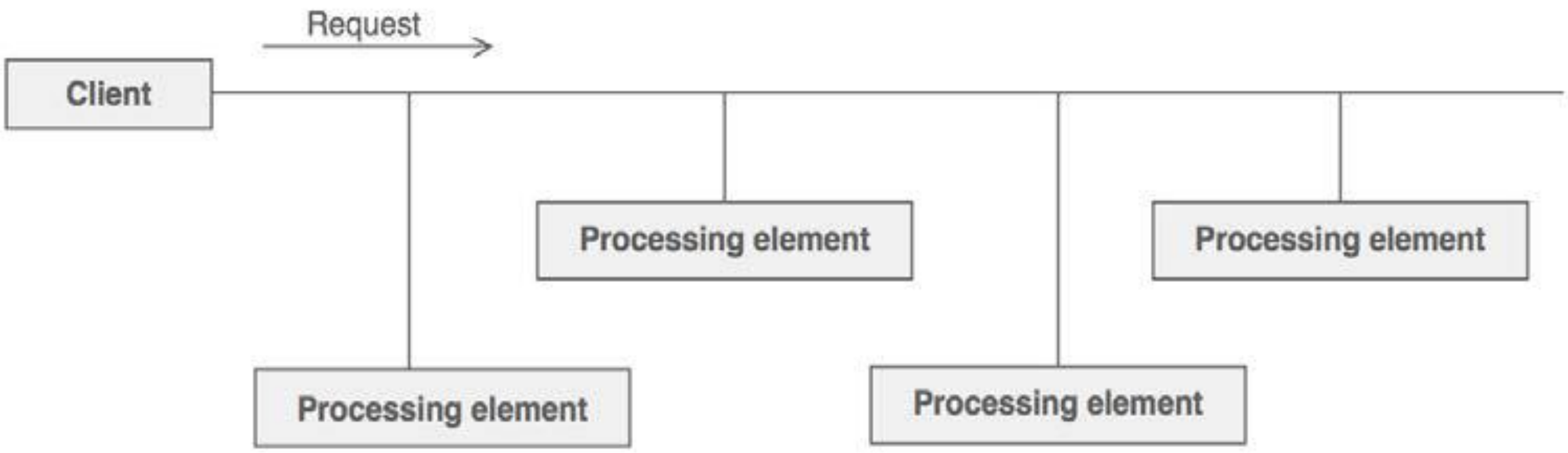
The Model-View-Controller Pattern

- A view is considered dumb because it:
- Displays the data
- Allows the user to interact with it
- Does only minimal processing, usually provided by a template language (for example, using simple variables and loop controls)
- Does not store any data
- Does not access the application data directly
- Does not contain validation/business rules/logic

The Chain of Responsibility Pattern

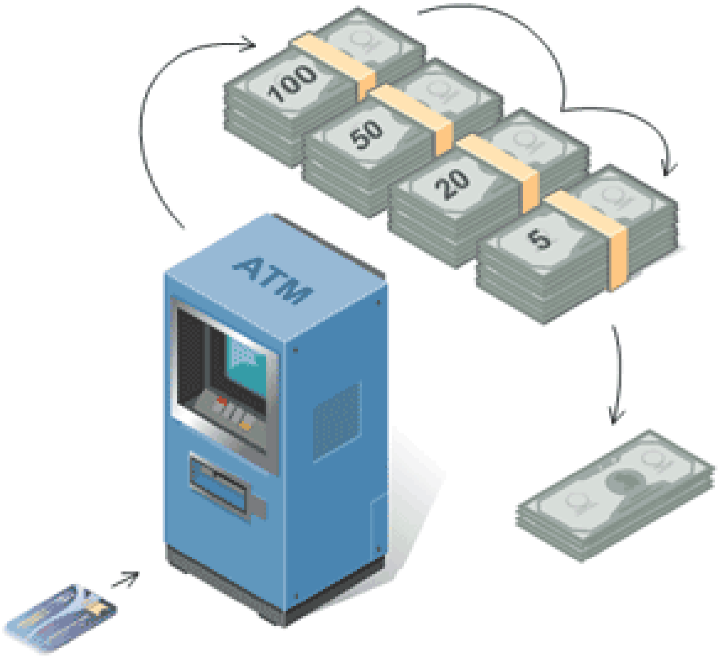
- The **Chain of Responsibility** pattern is used when we want to give a chance to multiple objects to satisfy a single request, or when we don't know which object (from a chain of objects) should process a specific request in advance.
- The principle is the same as the following:
- There is a chain (linked list, tree, or any other convenient data structure) of objects.
- We start by sending a request to the first object in the chain.
- The object decides whether it should satisfy the request or not.
- The object forwards the request to the next object.
- This procedure is repeated until we reach the end of the chain.

The Chain of Responsibility Pattern



The Chain of Responsibility Pattern

ATMs and, in general, any kind of machine that accepts/returns banknotes or coins (for example, a snack vending machine) use the chain of responsibility pattern. There is always a single slot for all banknotes, as shown in the following figure



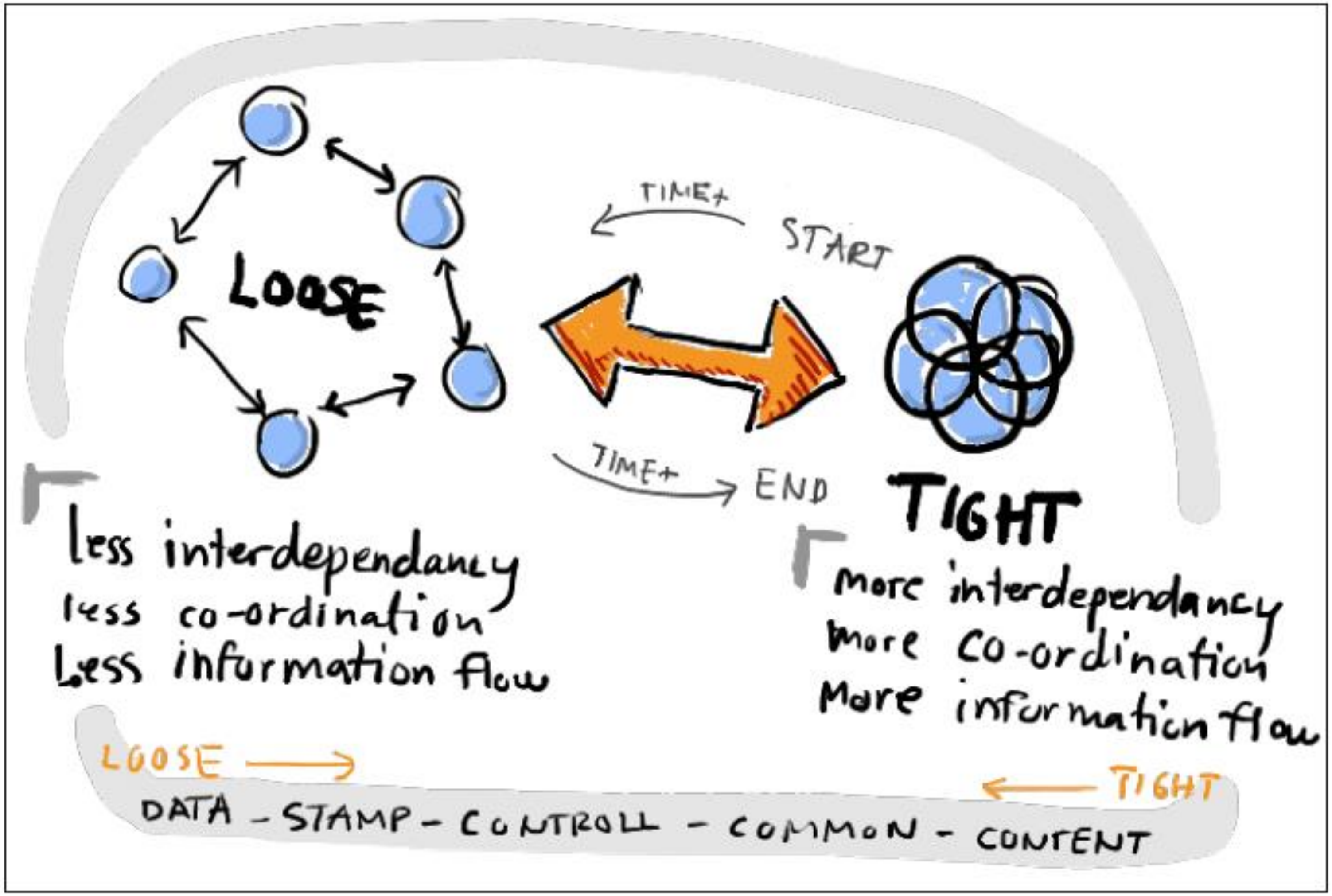
The Chain of Responsibility Pattern usecase

In purchase systems, there are many approval authorities. One approval authority might be able to approve orders up to a certain value, let's say $100.

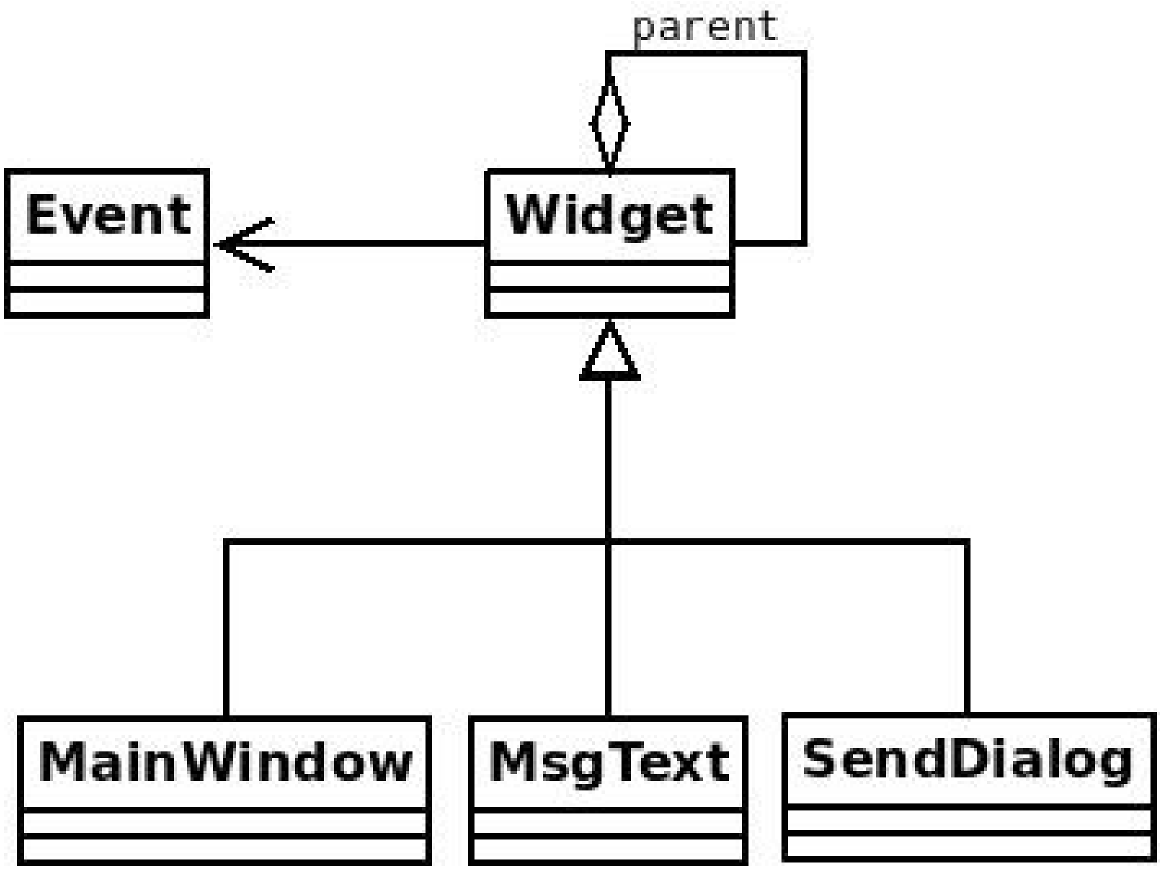
If the order is more than $100, the order is sent to the next approval authority in the chain that can approve orders up to $200, and so forth.

Another case where Chain of Responsibility is useful is when we know that more than one object might need to process a single request. This is what happens in an event-based programming. A single event such as a left mouse click can be caught by more than one listener.

The Chain of Responsibility Pattern usecase



The Chain of Responsibility Pattern usecase



Command Pattern

- The Command design pattern helps us encapsulate an operation (undo, redo, copy, paste, and so forth) as an object.
- What this simply means is that we create a class that contains all the logic and the methods required to implement the operation.

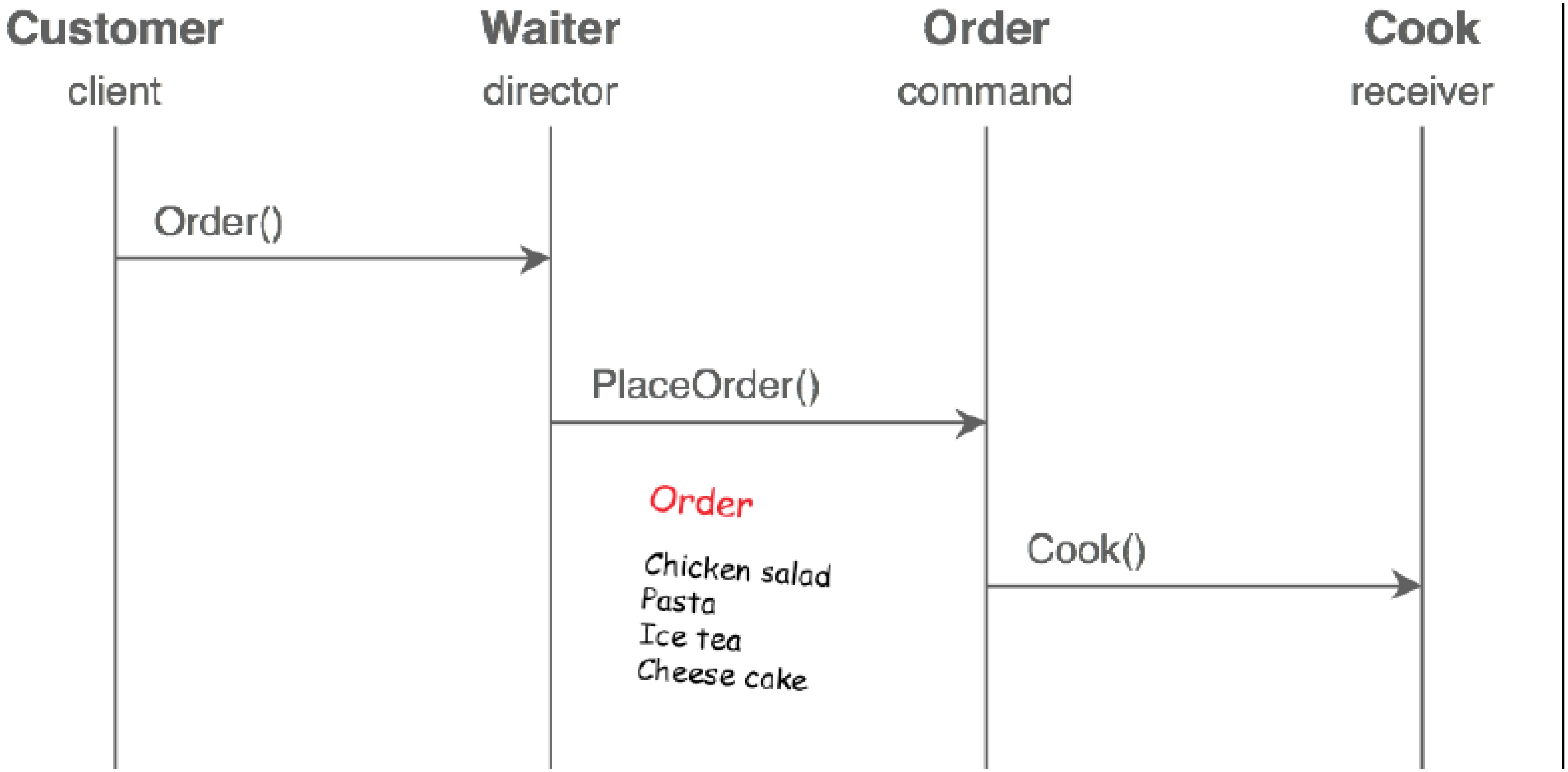
Command Pattern

- We don't have to execute a command directly. It can be executed on will.
- The object that invokes the command is decoupled from the object that knows how to perform it. The invoker does not need to know any implementation details about the command.
- If it makes sense, multiple commands can be grouped to allow the invoker to execute them in order. This is useful, for instance, when implementing a multilevel undo command.

Command Pattern Real Time

- When we go to the restaurant for dinner, we give the order to the waiter.
- The check (usually paper) they use to write the order on is an example of Command.
- After writing the order, the waiter places it in the check queue that is executed by the cook.
- Each check is independent and can be used to execute many and different commands, for example, one command for each item that will be cooked.

Command Pattern Real Time



Command Pattern Real Time – Use Case

- **GUI buttons and menu items**: The PyQt example that was already mentioned uses the Command pattern to implement actions on buttons and menu items.
- **Other operations**: Apart from undo, Command can be used to implement any operation. A few examples are cut, copy, paste, redo, and capitalize text.
- **Transactional behavior and logging**: Transactional behavior and logging are important to keep a persistent log of changes. They are used by operating systems to recover from system crashes, relational databases to implement transactions, filesystems to implement snapshots, and installers (wizards) to revert cancelled installations.
- **Macros**: By macros, in this case, we mean a sequence of actions that can be recorded and executed on demand at any point in time. Popular editors such as Emacs and Vim support macros.

Interpreter Pattern

- The **Interpreter** pattern is interesting only for the advanced users of an application.
- That's because the main idea behind Interpreter is to give the ability to non-beginner users and domain experts to use a simple language to express their ideas.[DSL]
- A DSL is a computer language of limited expressiveness targeting a particular domain.
- DSLs are used for different things, such as combat simulation, billing, visualization, configuration, communication protocols, and so on.
- DSLs are divided into internal DSLs and external DSLs

Interpreter Pattern

- Internal DSLs are built on top of a host programming language.
- An example of an internal DSL is a language that solves linear equations using Python.
- The advantages of using an internal DSL are that we don't have to worry about creating, compiling, and parsing grammar because these are already taken care of by the host language.
- The disadvantage is that we are constrained by the features of the host language. It is very challenging to create an expressive, concise, and fluent internal DSL if the host language does not have these features.

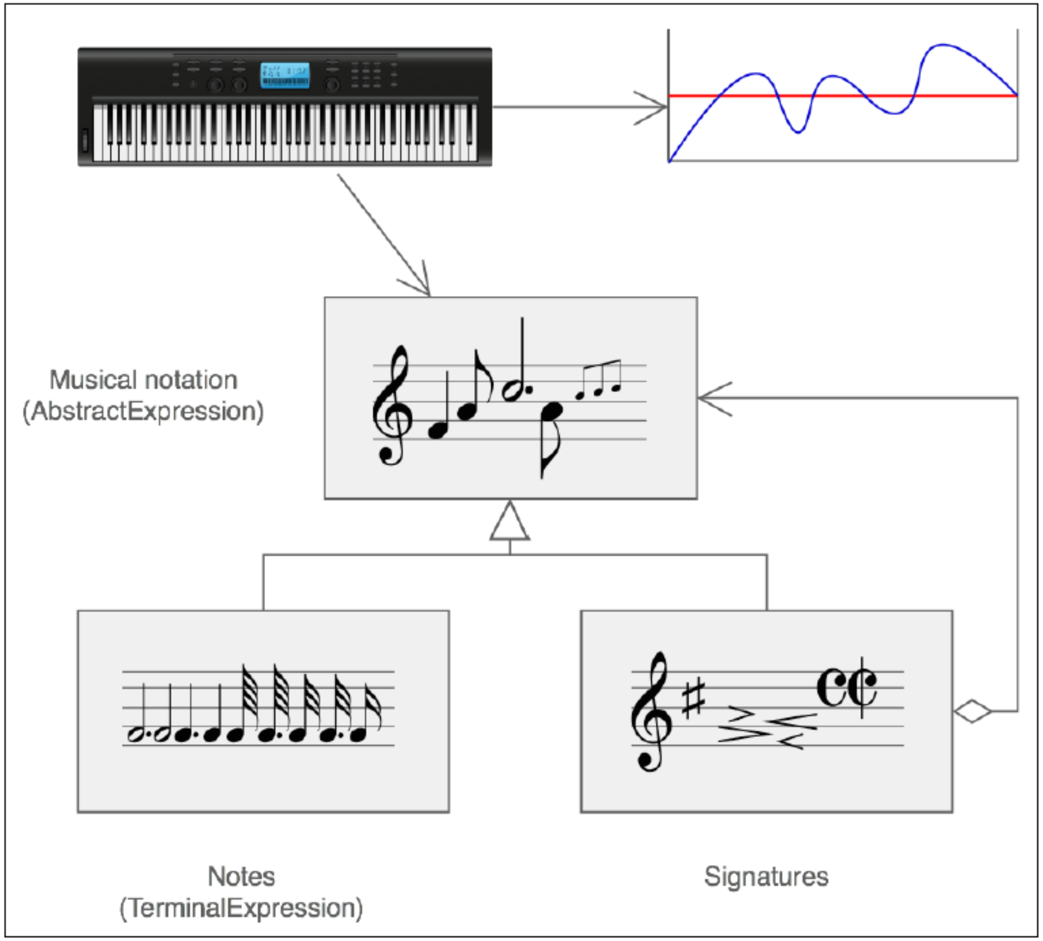
Interpreter Pattern

- External DSLs do not depend on host languages.
- The creator of the DSL can decide all aspects of the language (grammar, syntax, and so forth), but they are also responsible for creating a parser and compiler for it.
- Creating a parser and compiler for a new language can be a very complex, long, and painful procedure

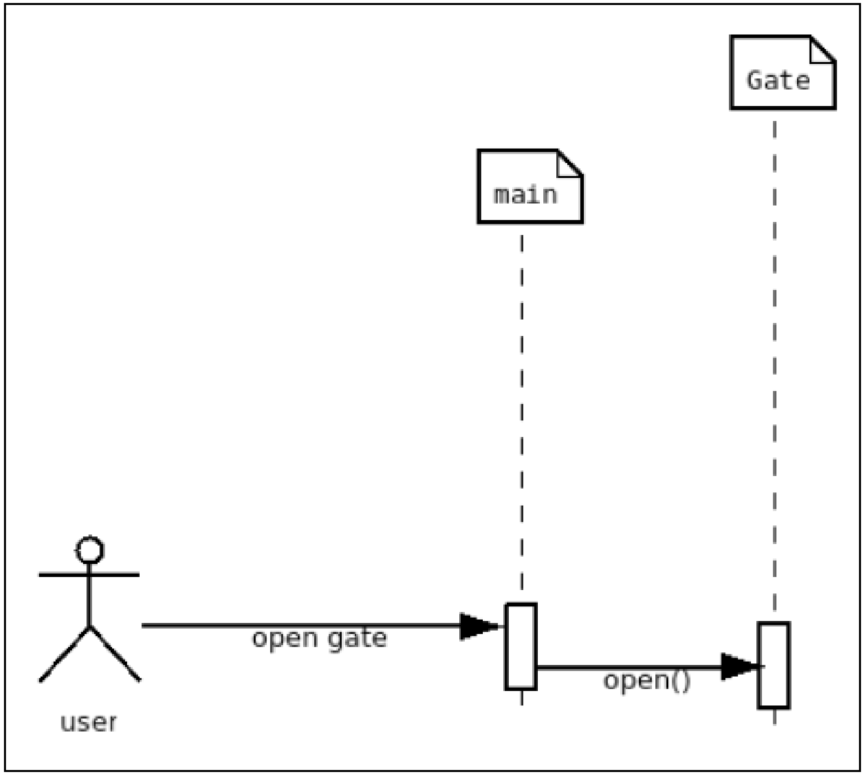
Interpreter Pattern

- A musician is an example of the Interpreter pattern in reality.
- Musical notation represents the pitch and duration of a sound graphically.
- The musician is able to reproduce a sound precisely based on its notation.
- In a sense, musical notation is the language of music, and the musician is the interpreter of that language.

Interpreter Pattern



Interpreter Pattern



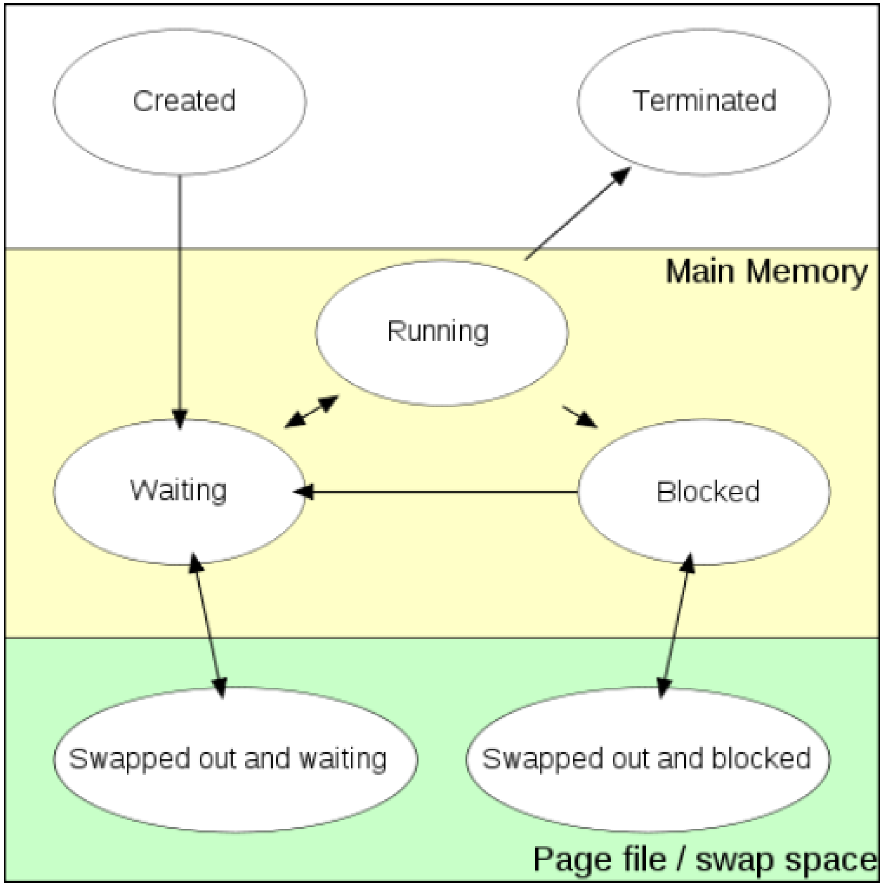
State Pattern

- A state machine is an abstract machine that has two key components: states and transitions.
- When a process is initially created by a user, it goes into the *created/new* state.
- From this state, the only transition is to go into the *waiting* state, which happens when the scheduler loads the process in memory and adds it to the queue of the processes that are *waiting/ready for execution*.
- A *waiting* process has two possible transitions: it can either be picked for execution (transition to *running*), or it can be replaced with a process that has higher priority (transition to *swapped out and waiting*).

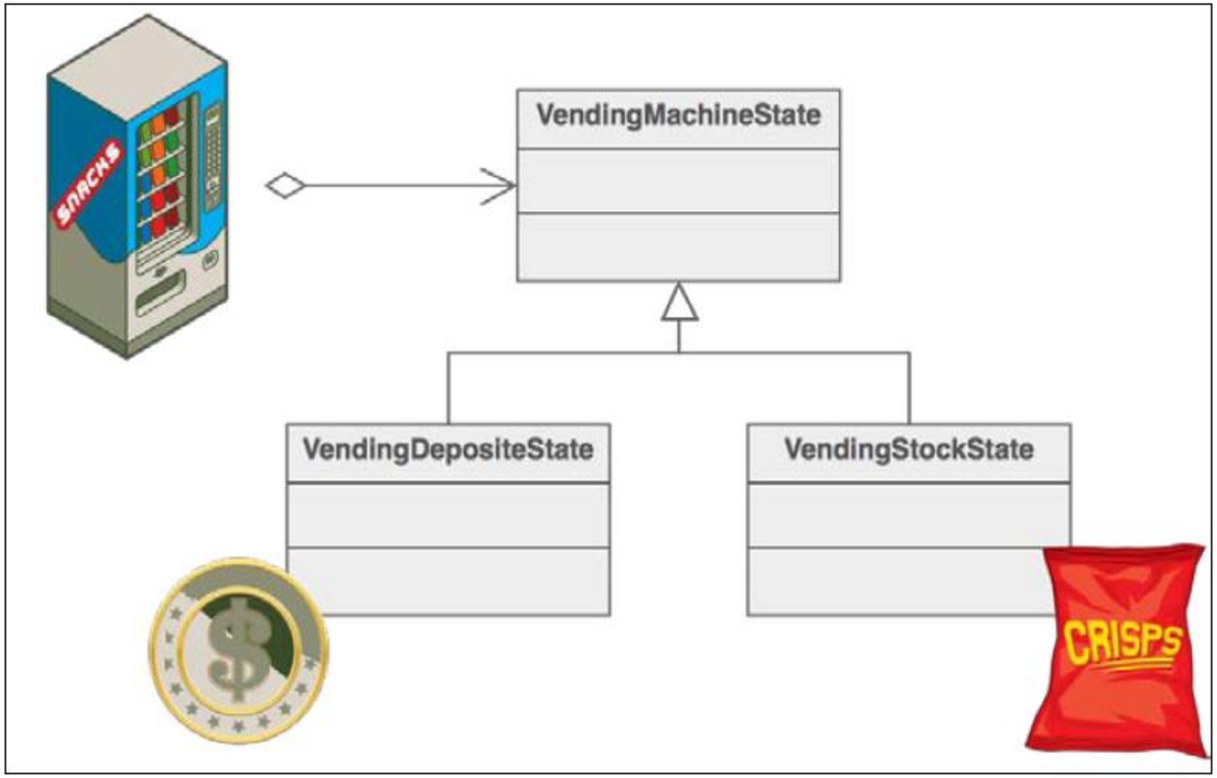
State Pattern

- Other typical states of a process are *terminated* (completed or killed), *blocked* (for example, waiting for an I/O operation to complete), and so forth. It is important to note that a state machine has only one active state at a specific point in time. For instance, a process cannot be at the same time in the state *created* and the state *running*.

State Pattern



State Pattern



State Pattern

- State machines are used to solve many computational and non-computational problems.
- Some of them are traffic lights, parking meters, hardware design, programming language parsing, and so forth. We saw how a snack vending machine relates to the way a state machine works.
- Modern software offers libraries/modules to make the implementation and usage of state machines easier.
- Django offers the third-party django-fsm package and Python also has many contributed modules.
- In fact, one of them (state_machine) was used in the implementation section. The **State Machine Compiler** (**SMC**) is yet another promising project, offering many programming language bindings (including Python).

The Strategy Pattern

- Most problems can be solved in more than one way. Take, for example, the sorting problem, which is related to putting the elements of a list in a specific order.
- There are many sorting algorithms, and, in general, none of them is considered the best for all cases
- There are different criteria that help us pick a sorting algorithm on a per-case basis.

The Strategy Pattern

- **Number of elements that need to be sorted**: This is called the input size. Almost all the sorting algorithms behave fairly well when the input size is small, but only a few of them have good performance with a large input size.
- **Best/average/worst time complexity of the algorithm**: Time complexity is (roughly) the amount of time the algorithm takes to complete, excluding coefficients and lower order terms. This is often the most usual criterion to pick an algorithm, although it is not always sufficient.

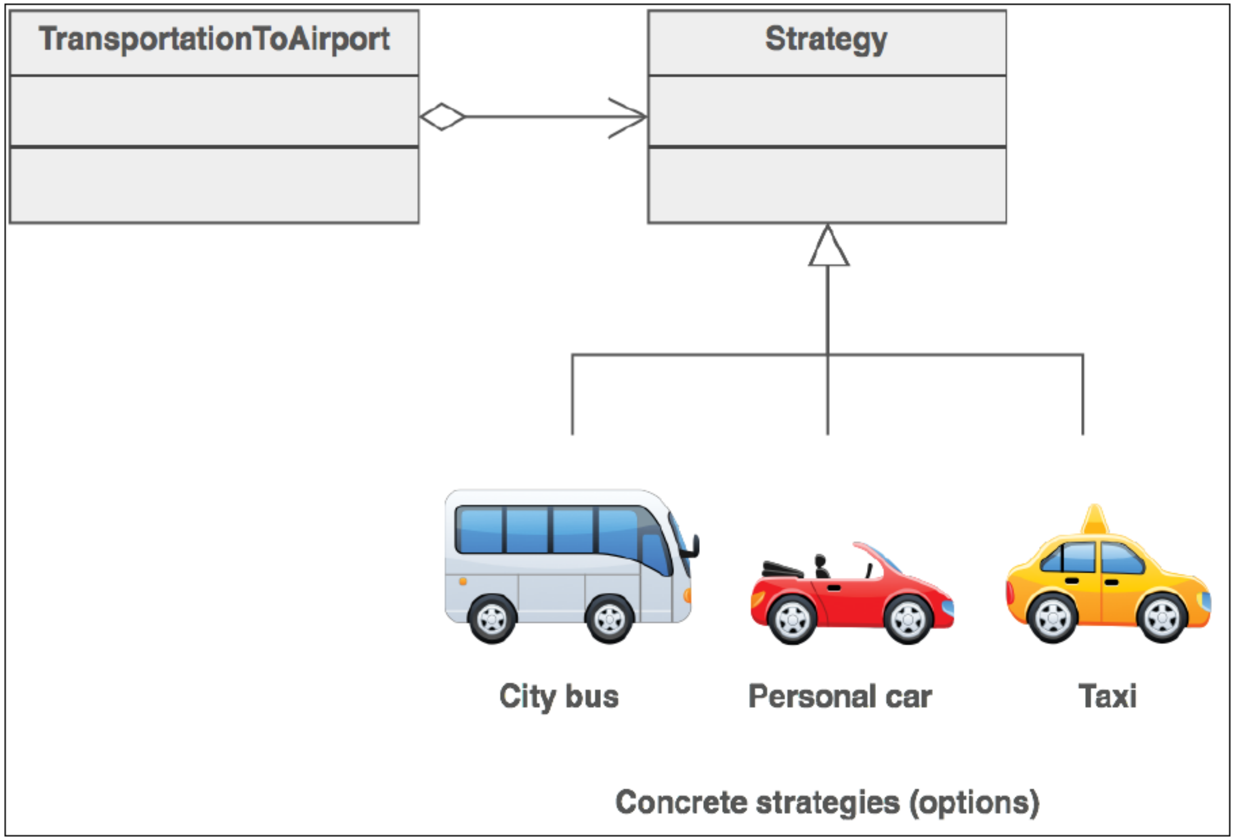
The Strategy Pattern

- **Space complexity of the algorithm**: Space complexity is (again roughly) the amount of physical memory needed to fully execute an algorithm. This is very important when we are working with big data or embedded systems, which usually have limited memory.
- **Stability of the algorithm**: An algorithm is considered stable when it maintains the relative order of elements with equal values after it is executed.

The Strategy Pattern

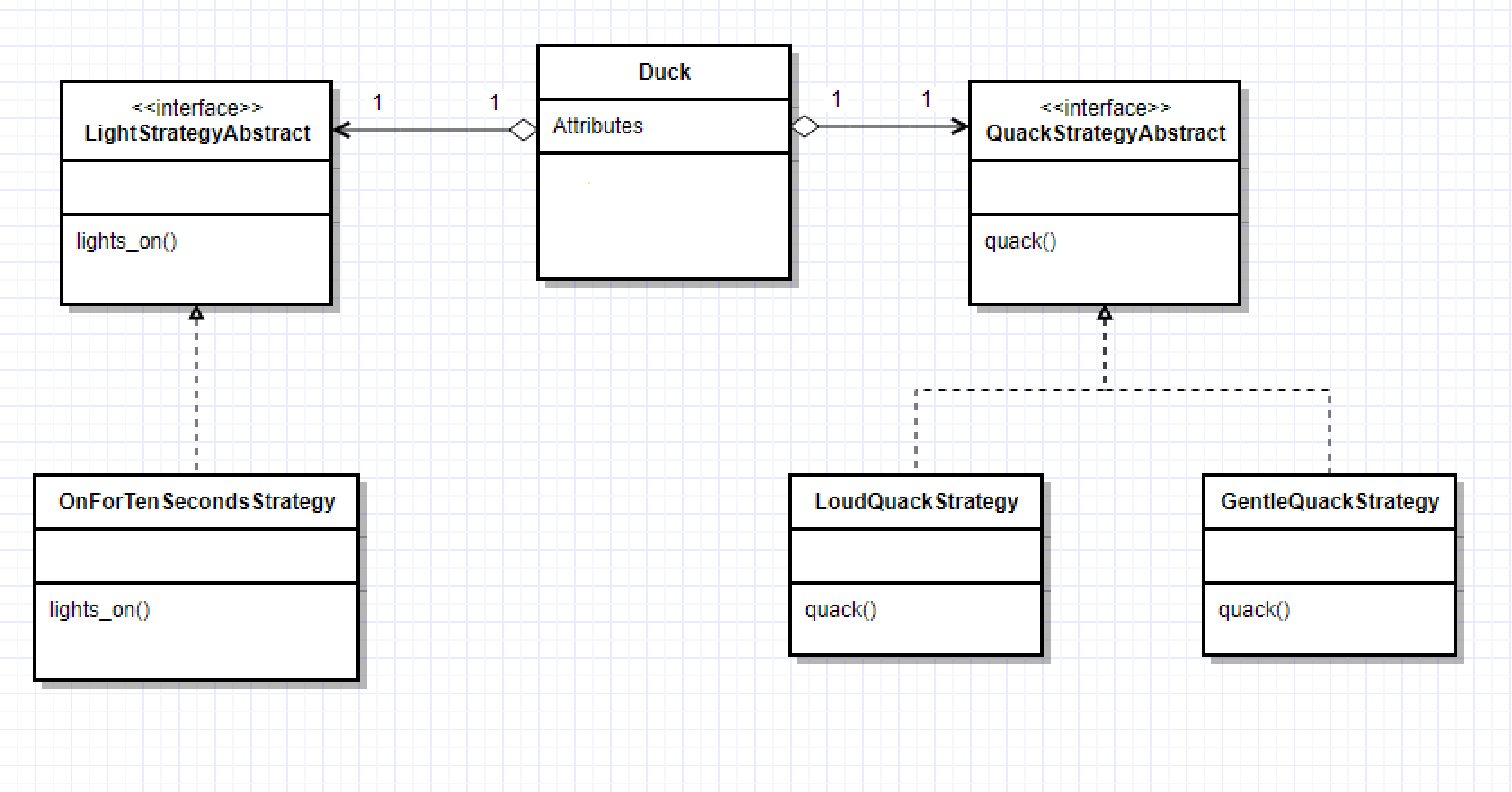
- **Code complexity of the algorithm**: If two algorithms have the same time/space complexity and are both stable, it is important to know which algorithm is easier to code and maintain.

The Strategy Pattern



The Strategy Pattern

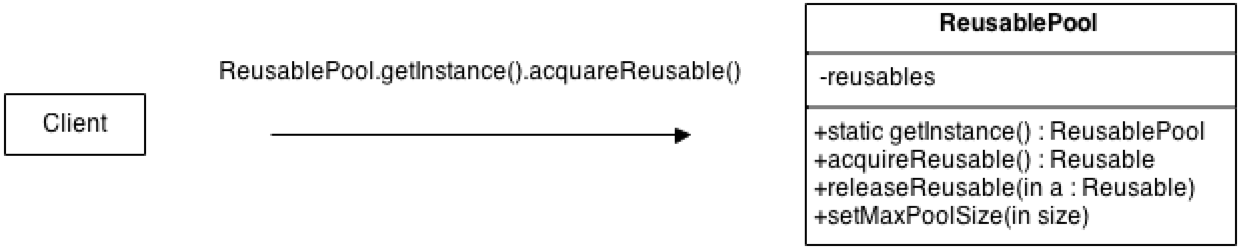
The Strategy Pattern



Object Pool Design Pattern

- Object pooling can offer a significant performance boost;
- it is most effective in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instantiations in use at any one time is low.
- Object pools (otherwise known as resource pools) are used to manage the object caching.
- A client with access to a Object pool can avoid creating a new Objects by simply asking the pool for one that has already been instantiated instead.
- Generally the pool will be a growing pool, i.e. the pool itself will create new objects if the pool is empty, or we can have a pool, which restricts the number of objects created.

Object Pool Design Pattern



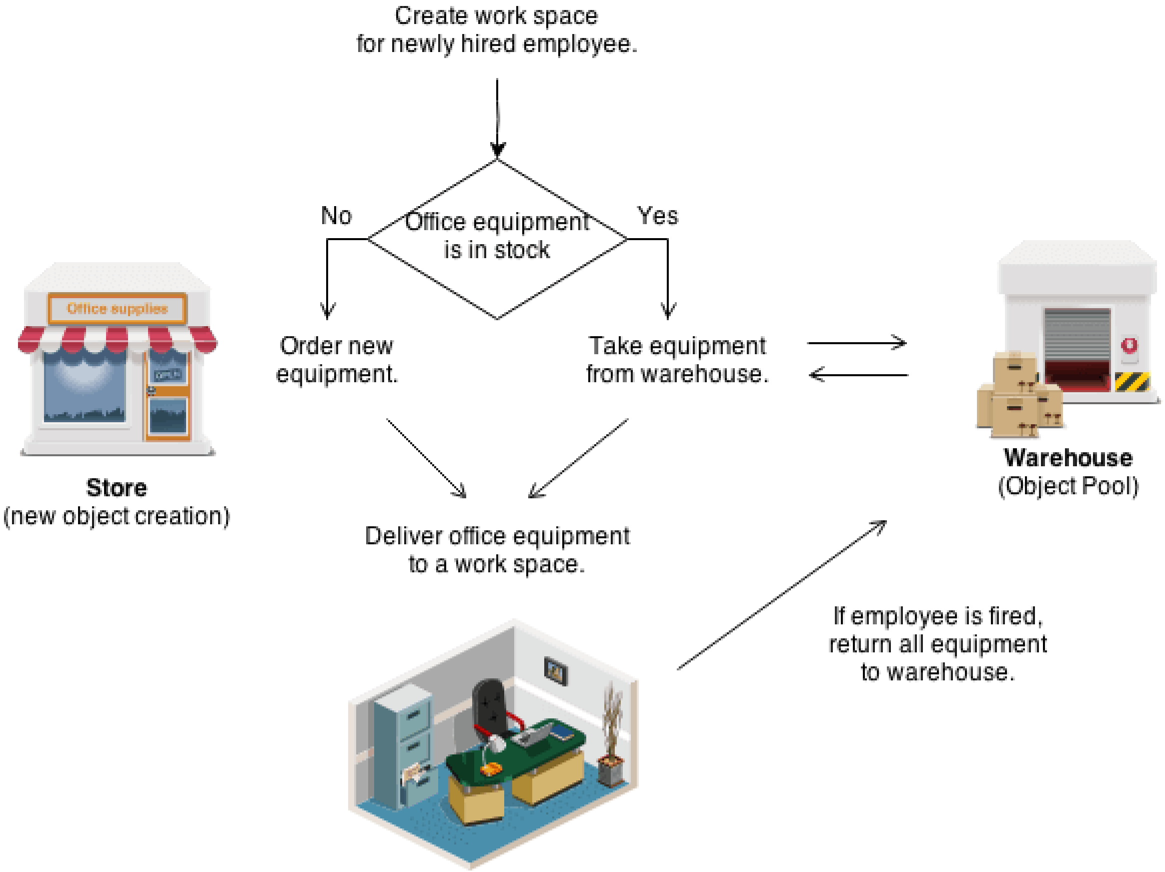
Client

ReusablePool.getInstance().acquareReusable()

ReusablePool

-reusables

+static getInstance() : ReusablePool
+acquireReusable() : Reusable
+releaseReusable(in a : Reusable)
+setMaxPoolSize(in size)



Create work space
for newly hired employee.

Office equipment
is in stock

No

Yes

Order new
equipment.

Take equipment
from warehouse.

Deliver office equipment
to a work space.

If employee is fired,
return all equipment
to warehouse.

Store
(new object creation)

Warehouse
(Object Pool)

Template Pattern

- The Template Method pattern is a behavioral design pattern that defines the program skeleton or an algorithm in a method called the Template Method.
- For example, you could define the steps to prepare a beverage as an algorithm in a Template Method.
- The Template Method pattern also helps redefine or customize certain steps of the algorithm by deferring the implementation of some of these steps to subclasses.
- This means that the subclasses can redefine their own behavior.

Template Pattern

- For example, in this case, subclasses can implement steps to prepare tea using the Template Method to prepare a beverage.
- It is important to note that the change in the steps (as done by the subclasses) don't impact the original algorithm's structure.
- Thus, the facility of overriding by subclasses in the Template Method pattern allows the creation of different behaviors or algorithms.

Template Pattern

- Abstract class is used to define the steps of the algorithm.
- These steps are also known as *primitive operations* in the context of the Template Method pattern.
- These steps are defined with abstract methods, and the Template Method defines the algorithm.
- The ConcreteClass (that subclasses the abstract class) implements subclass-specific steps of the algorithm.

Template Pattern

- The Template Method pattern is used in the following cases:
- When multiple algorithms or classes implement similar or identical logic
- The implementation of algorithms in subclasses helps reduce code duplication
- Multiple algorithms can be defined by letting the subclasses implement the behavior through overriding

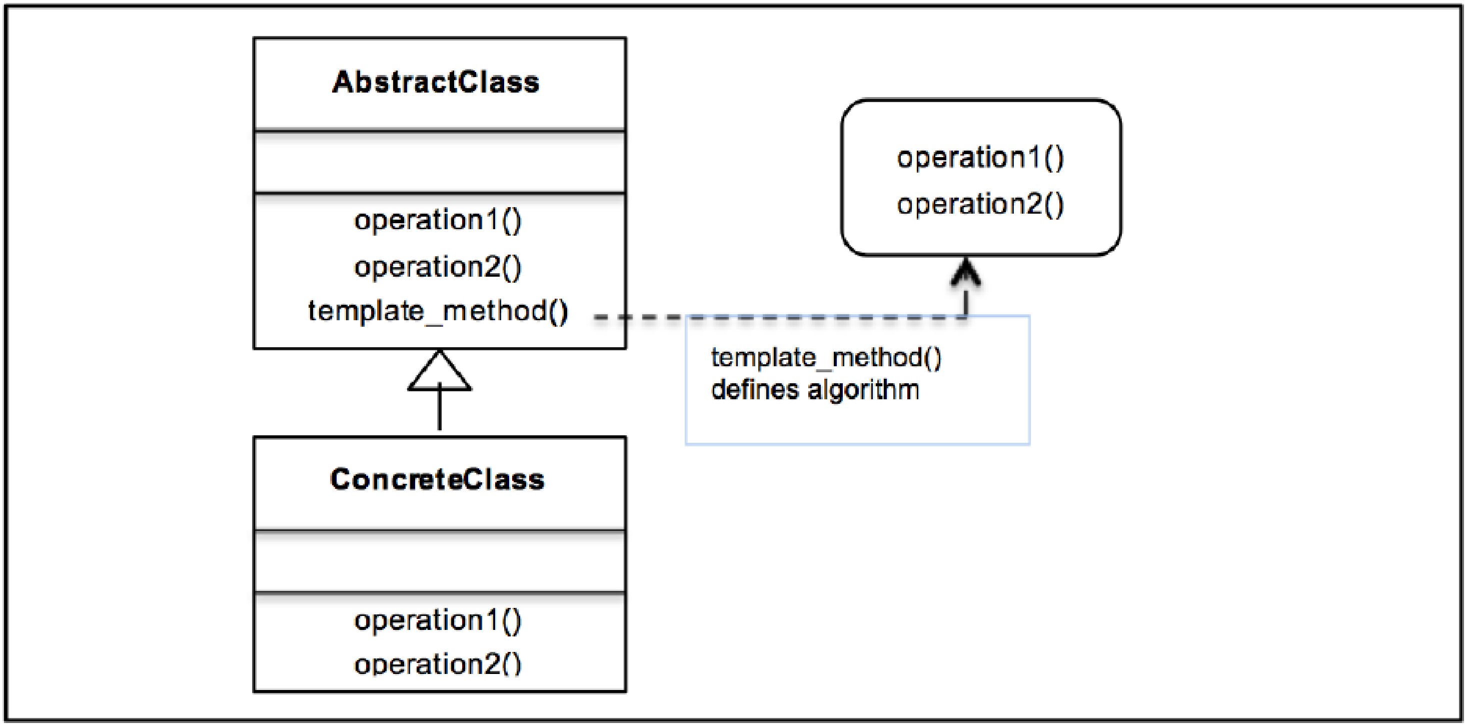
Template Pattern

- In short, the main intentions of the Template Method pattern are as follows:
- Defining a skeleton of an algorithm with primitive operations
- Redefining certain operations of the subclass without changing the algorithm's structure
- Achieving code reuse and avoiding duplicate efforts
- Leveraging common interfaces or implementations

Template Pattern

- The Template Method pattern works with the following terms—AbstractClass, ConcreteClass, Template Method, and Client:
- AbstractClass: This declares an interface to define the steps of the algorithm
- ConcreteClass: This defines subclass-specific step definitions
- template_method(): This defines the algorithm by calling the step methods

Template Pattern



Template Pattern

- Imagine the case of a travel agency, say, Dev Travels. Now how do they typically work?
- They define various trips to various locations and come up with a holiday package for you.
- A package is essentially a trip that you, as a customer, undertakes.
- A trip has details such as the places visited, transportation used, and other factors that define the trip itinerary.
- This same trip can be customized differently based on the needs of the customers

Template Pattern

- The Template Method pattern provides you with the following advantages:
- Code reuse happens with the Template Method pattern as it uses inheritance and not composition. Only a few methods need to be overridden.
- Flexibility lets subclasses decide how to implement steps in an algorithm.

Template Pattern

- Debugging and understanding the sequence of flow in the Template Method pattern can be confusing at times.
- You may end up implementing a method that shouldn't be implemented or not implementing an abstract method at all.
- Documentation and strict error handling has to be done by the programmer.
- Maintenance of the template framework can be a problem as changes at any level (low-level or high-level) can disturb the implementation. Hence, maintenance can be painful with the Template Method pattern.

Bridge Pattern

- The Bridge design pattern allows you to separate the abstraction from the implementation.
- It is a structural design pattern.
- **There are 2 parts in Bridge design pattern :**
  - Abstraction
  - Implementation
- **Bridge pattern is about preferring composition over inheritance. Implementation details are pushed from a hierarchy to another object with a separate hierarchy**.

Bridge Pattern

- The bridge pattern allows the Abstraction and the Implementation to be developed independently and the client code can access only the Abstraction part without being concerned about the Implementation part.
- The abstraction is an interface or abstract class and the implementor is also an interface or abstract class.
- The abstraction contains a reference to the implementor.
- Children of the abstraction are referred to as refined abstractions, and children of the implementor are concrete implementors.

Bridge Pattern

- Since we can change the reference to the implementor in the abstraction, we are able to change the abstraction’s implementor at run-time.
- Changes to the implementor do not affect client code.
- It increases the loose coupling between class abstraction and it’s implementation.
- *Decouple an abstraction from its implementation so that the two can vary independently.*

Bridge Pattern

- Let’s say that you want to build a news website and you want to show different content for different users.
- You want to give paid users full access to articles, without any ads.
- At the same time, you want to give free users some excerpts from the articles, with some ads on the page.
- Finally, you want to show a *call to action* to all free users, so you can hopefully convert them to paid users.

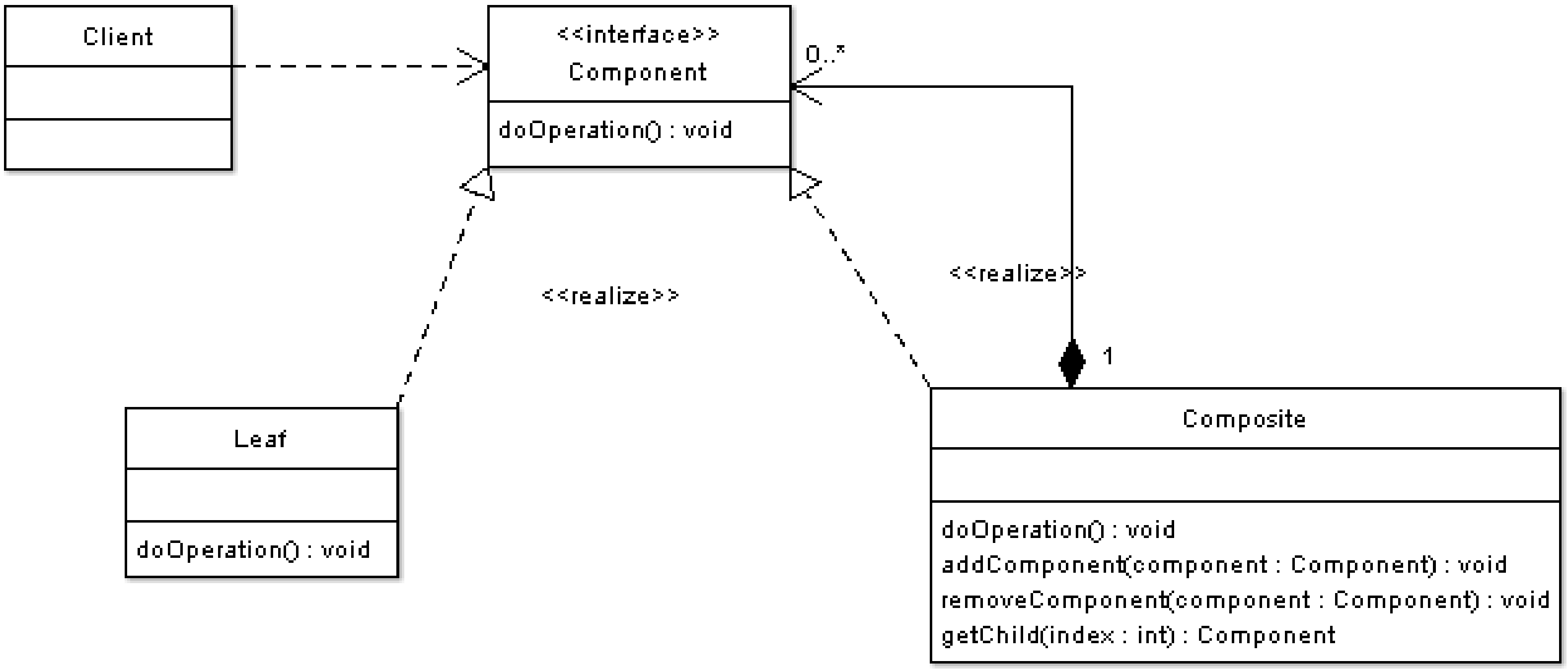
Composite Design Pattern

- **Composite** is a Conceptual design pattern that allows composing objects into a tree-like structure and work with the it as if it was a singular object.
- *Composite pattern lets clients to treat the individual objects in a uniform manner.*
- The Composite Pattern allows you to compose objects into a tree structure to represent the part-whole hierarchy which means you can create a tree of objects that is made of different parts, but that can be treated as a whole one big thing.
- Composite lets clients to treat individual objects and compositions of objects uniformly.

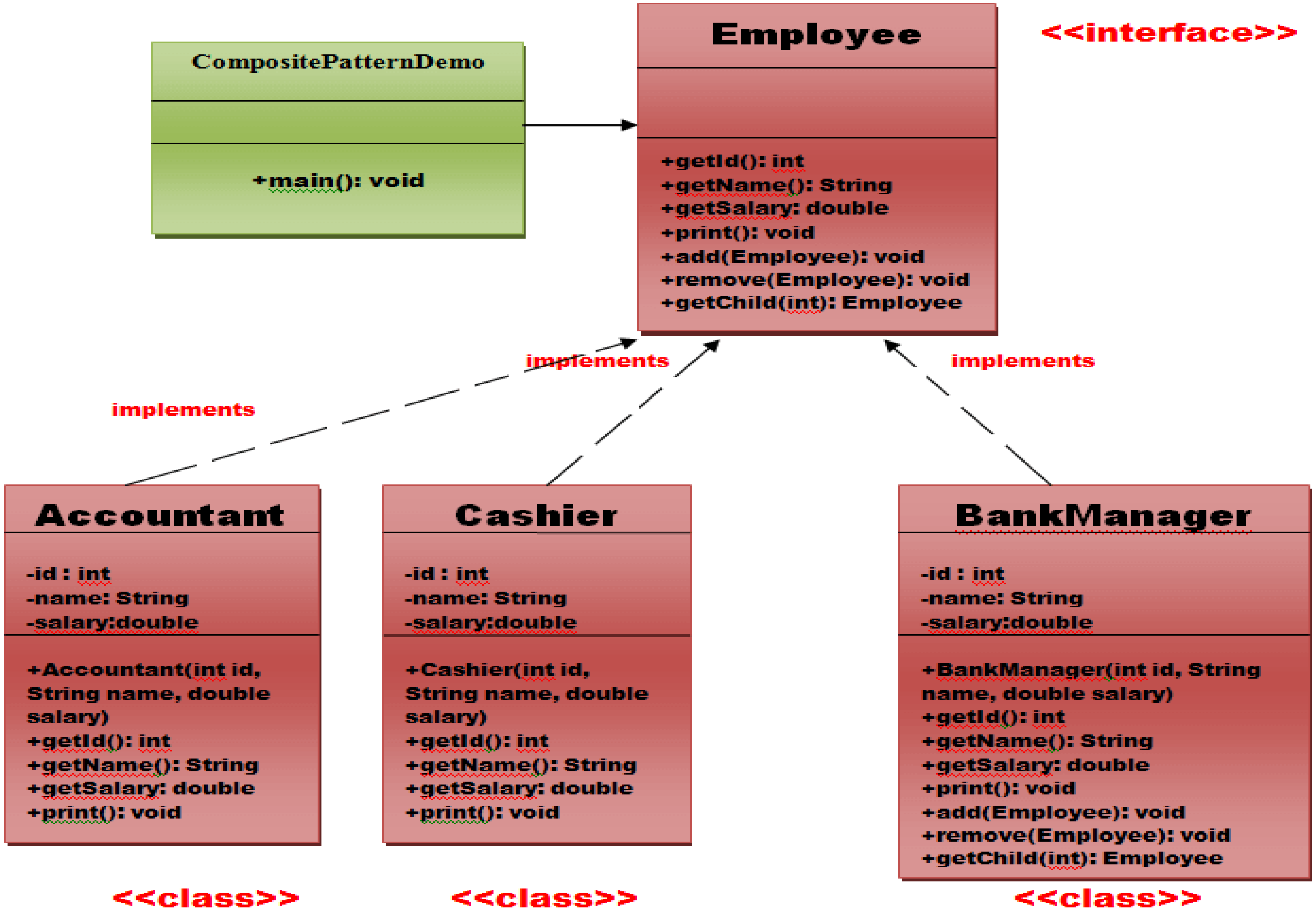
Composite Design Pattern

- **for example a program that manipulates a file system. A file system is a tree structure that contains Branches which are Folders as well as Leaf nodes which are Files.**
- **Note that a folder object usually contains one or more file or folder objects and thus is a complex object where a file is a simple object.**
- **Note also that since files and folders have many operations and attributes in common, such as moving and copying a file or a folder, listing file or folder attributes such as file name and size.**
- **It would be easier and more convenient to treat both file and folder objects uniformly by defining a File System Resource Interface**

Composite Design Pattern



Composite Design Pattern

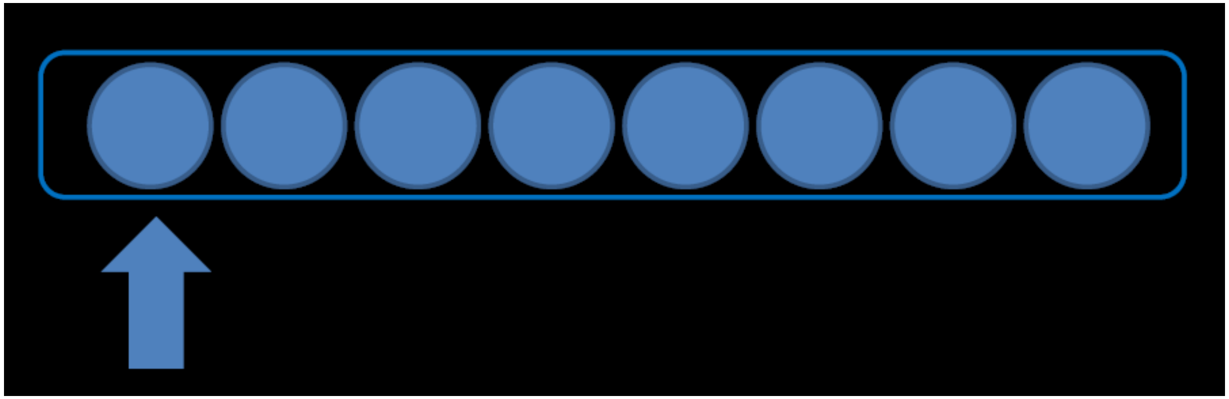


Composite Design Pattern

- Composite Pattern important points
- Composite pattern should be applied only when the group of objects should behave as the single object.
- Composite design pattern can be used to create a tree like structure.

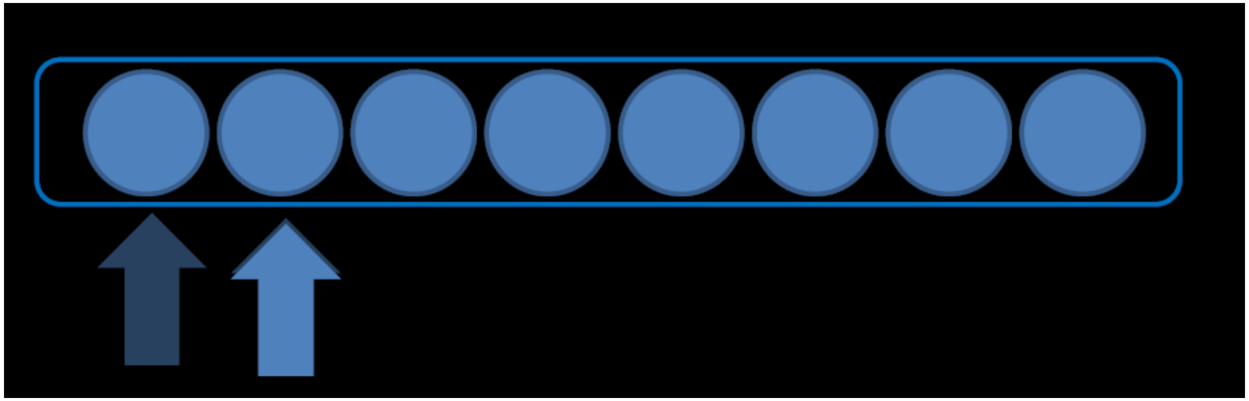
Iterator Pattern

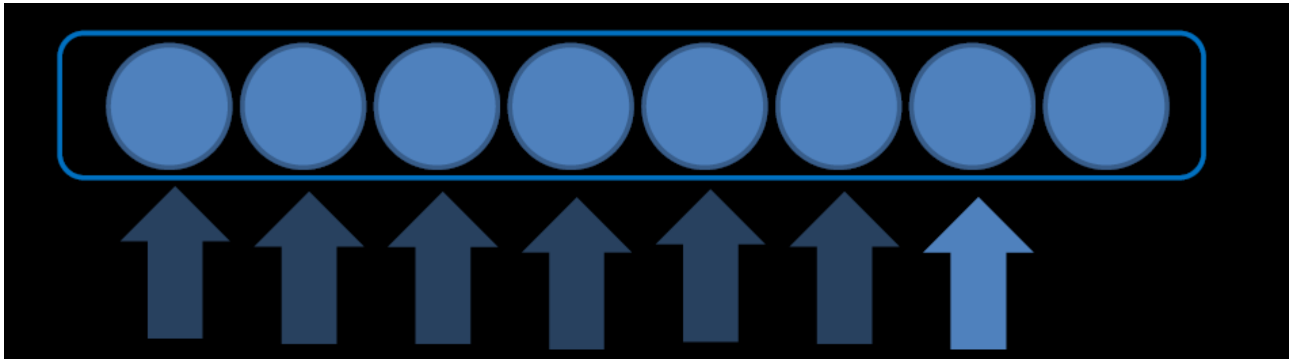
- The iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements.



Iterator Pattern

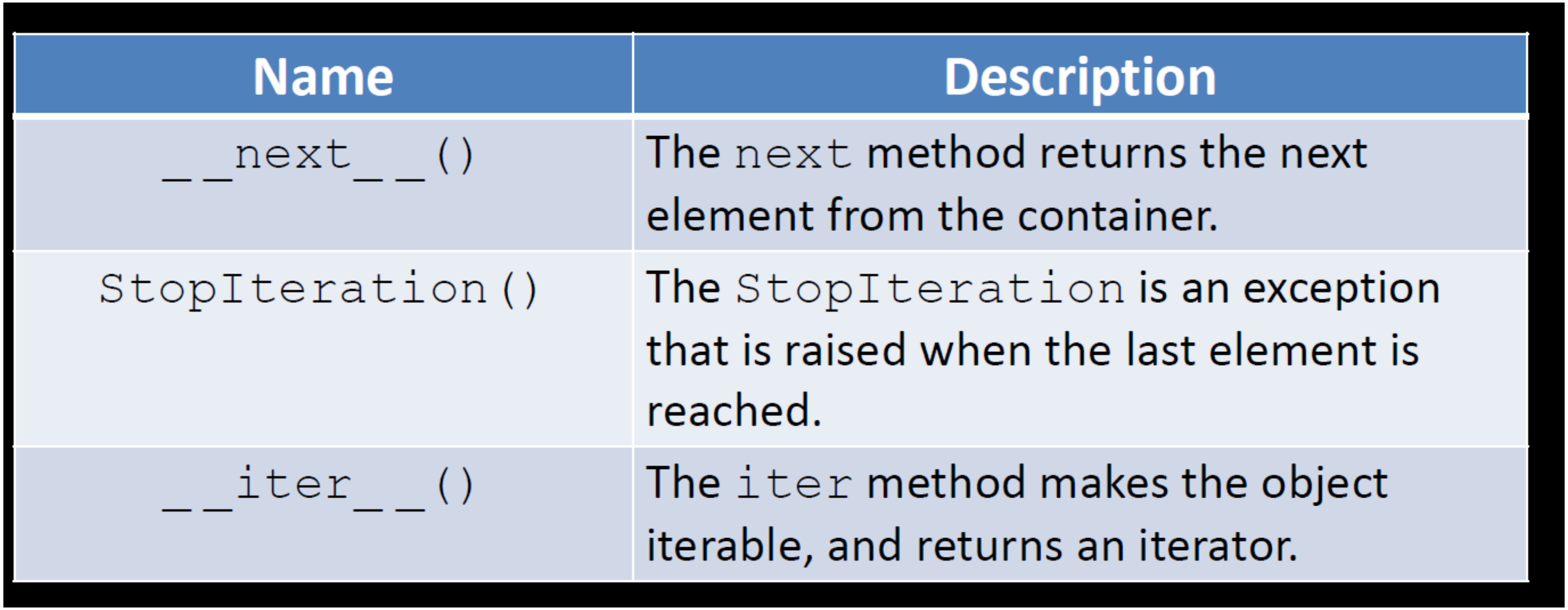
The iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements.





Iterator Pattern

In a programming language without patterns, an iterator would have a next() method and a done() method, and the iterator loops across all the containers using these methods.



Name	Description
`__next__()`	The `next` method returns the next element from the container.
`StopIteration()`	The `StopIteration` is an exception that is raised when the last element is reached.
`__iter__()`	The `iter` method makes the object iterable, and returns an iterator.

Iterator Pattern

The iterator pattern has two parts:

– An iterable class (to create the iterator object)

– An iterator class (to traverse a container)

Iterator Pattern

Advantage of Iterator Pattern

- It supports variations in the traversal of a collection.
- It simplifies the interface to the collection.

Usage of Iterator Pattern:

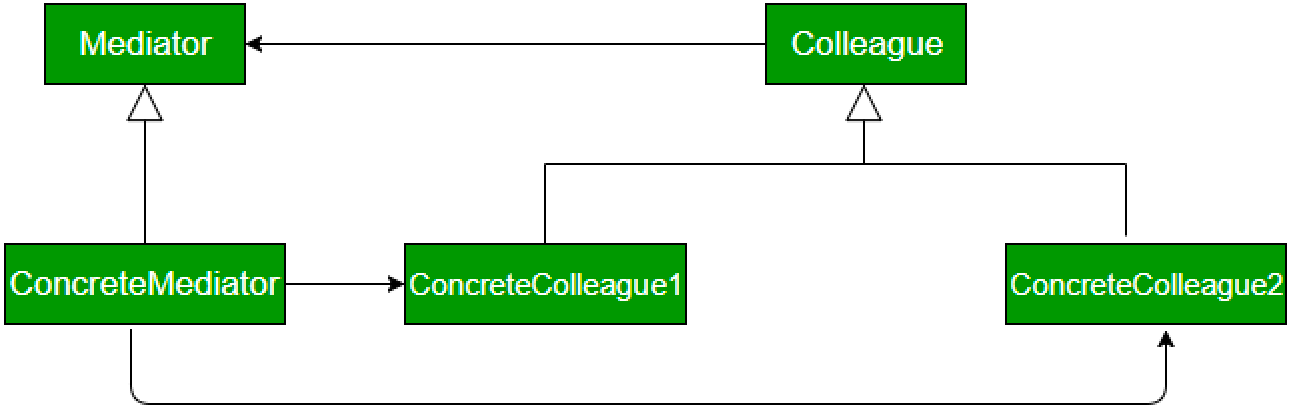
- It is used:
- When you want to access a collection of objects without exposing its internal representation.
- When there are multiple traversals of objects need to be supported in the collection.
-

Mediator Pattern

- Mediator design pattern is one of the important and widely used behavioral design pattern.
- Mediator enables decoupling of objects by introducing a layer in between so that the interaction between objects happen via the layer.
- If objects interact with each other directly, the system components are tightly-coupled with each other that makes higher maintainability cost and not hard to extend.
- Mediator pattern focuses on providing a mediator between objects for communication and help in implementing lose-coupling between objects.

Mediator Pattern

- Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights.
- Mediator works as a router between objects and it can have it’s own logic to provide way of communication.



Mediator Pattern

Advantage

- It limits subclassing. A mediator localizes behavior that otherwise would be distributed among several objects. Changing this behaviour requires subclassing Mediator only, Colleague classes can be reused as is.

Disadvantage

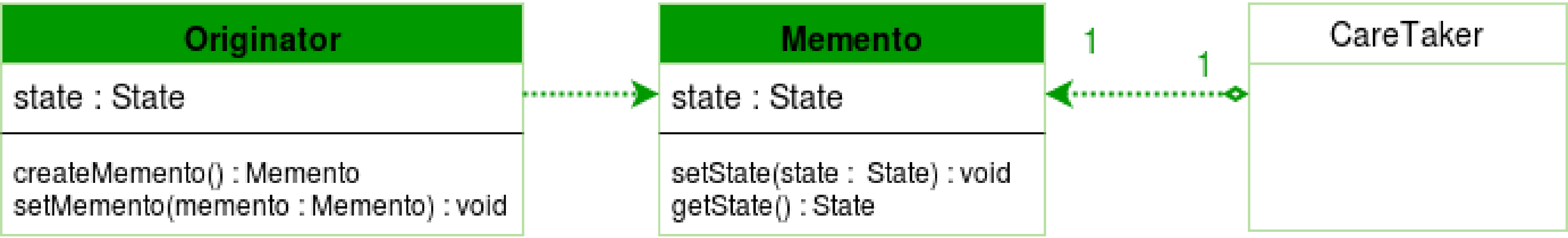
- It centralizes control. The mediator pattern trades complexity of interaction for complexity in the mediator. Because a mediator encapsulates protocols, it can become more complex than any individual colleague. This can make the mediator itself a monolith that's hard to maintain

Memento design pattern

- Memento pattern is a behavioral design pattern.
- Memento pattern is used to restore state of an object to a previous state.
- As your application is progressing, you may want to save checkpoints in your application and restore back to those checkpoints later.

Memento design pattern

- **originator :** the object for which the state is to be saved. It creates the memento and uses it in future to undo.
- **memento :** the object that is going to maintain the state of originator. Its just a POJO.
- **caretaker :** the object that keeps track of multiple memento. Like maintaining savepoints.


Originator
state : State
createMemento() : Memento
setMemento(memento : Memento) : void
Memento
state : State
setState(state : State) : void
getState() : State
1
1
CareTaker

Memento design pattern

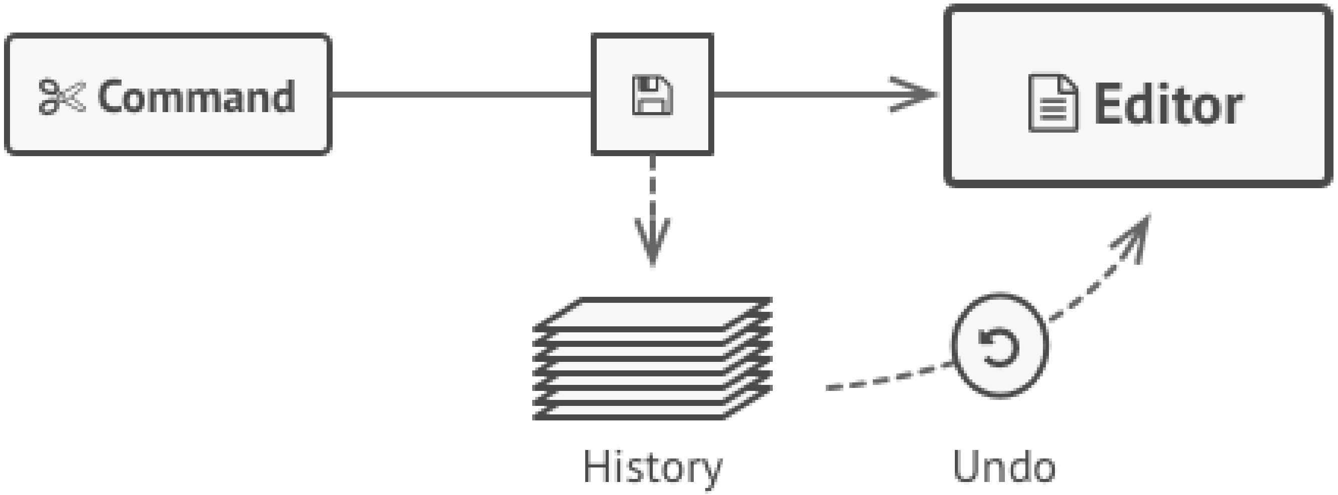
- A **Caretaker** would like to perform an operation on the **Originator** while having the possibility to rollback.
- The caretaker calls the **createMemento()** method on the originator asking the originator to pass it a memento object.
- At this point the originator creates a memento object saving its internal state and passes the memento to the caretaker.
- The caretaker maintains the memento object and performs the operation.
- In case of the need to undo the operation, the caretaker calls the **setMemento()** method on the originator passing the maintained memento object.
- The originator would accept the memento, using it to restore its previous state.

Memento design pattern

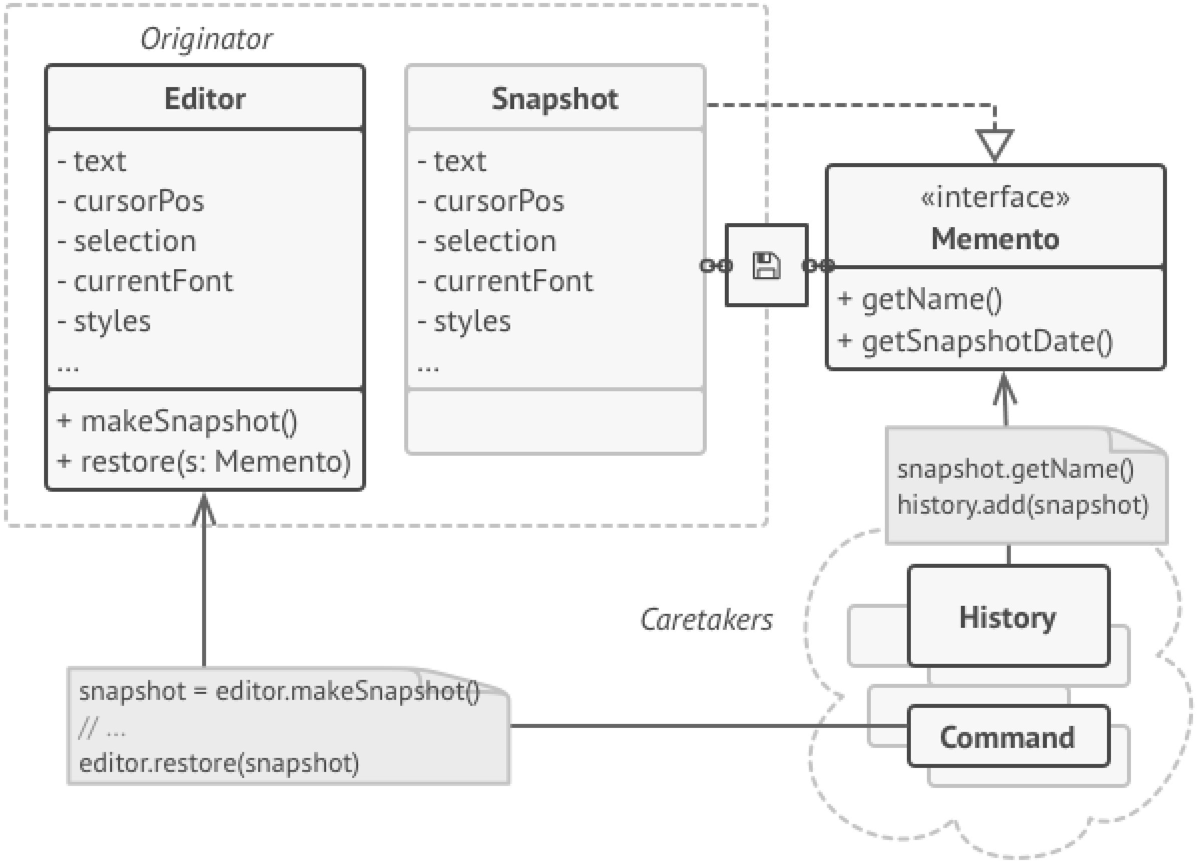
- **Advantage**
- We can use Serialization to achieve memento pattern implementation that is more generic rather than Memento pattern where every object needs to have it's own Memento class implementation.
- **Disadvantage**
- If Originator object is very huge then Memento object size will also be huge and use a lot of memory.

Memento design pattern

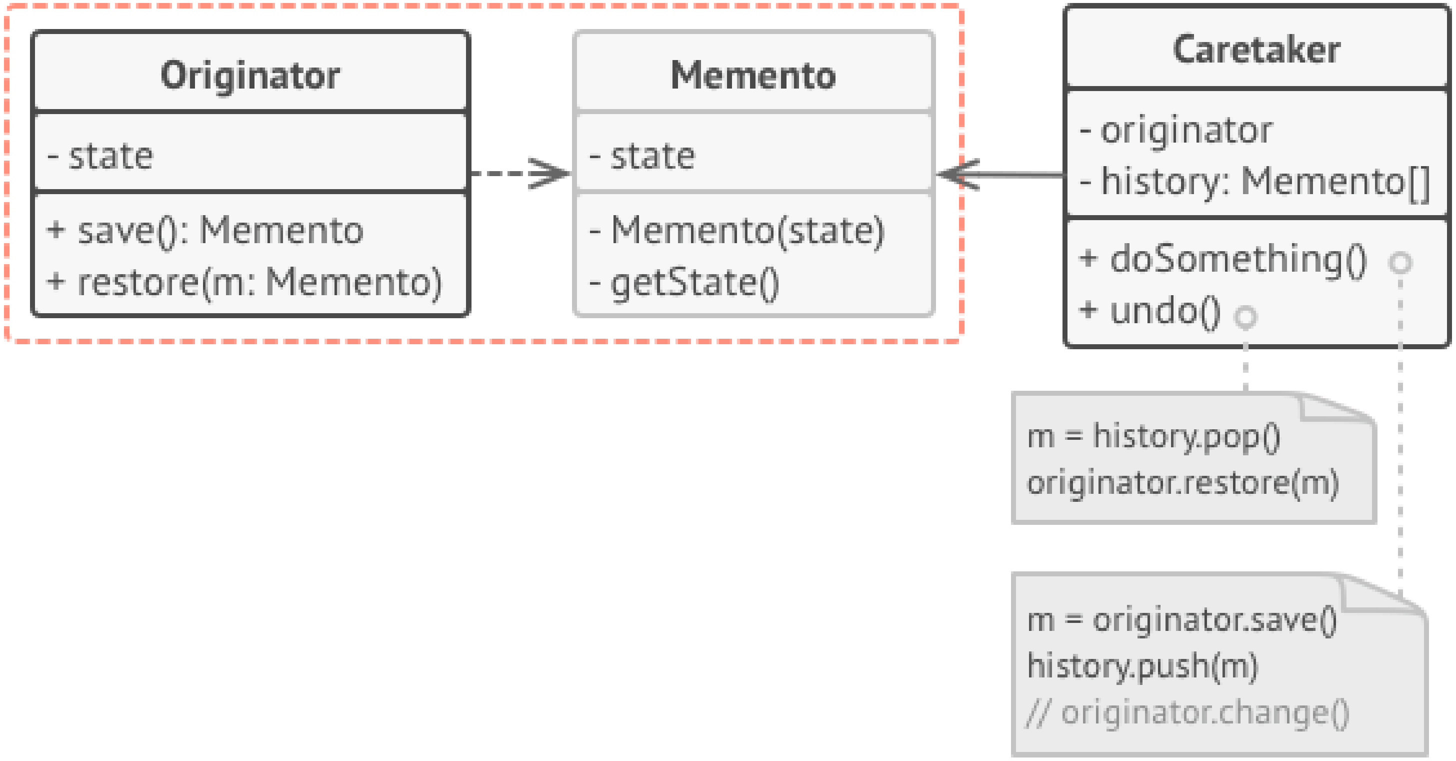
Memento design pattern



Memento design pattern



Memento design pattern



Originator

- state

+ save(): Memento
+ restore(m: Memento)

Memento

- state

- Memento(state)
- getState()

Caretaker

- originator
- history: Memento[]

+ doSomething()
+ undo()

```
m = history.pop()
originator.restore(m)
```

```
m = originator.save()
history.push(m)
// originator.change()
```

Memory Management

- Memory management in Python involves a private heap containing all Python objects and data structures.
- The management of this private heap is ensured internally by the *Python memory manager*.
- The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

Memory Management

- At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system.
- On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type.

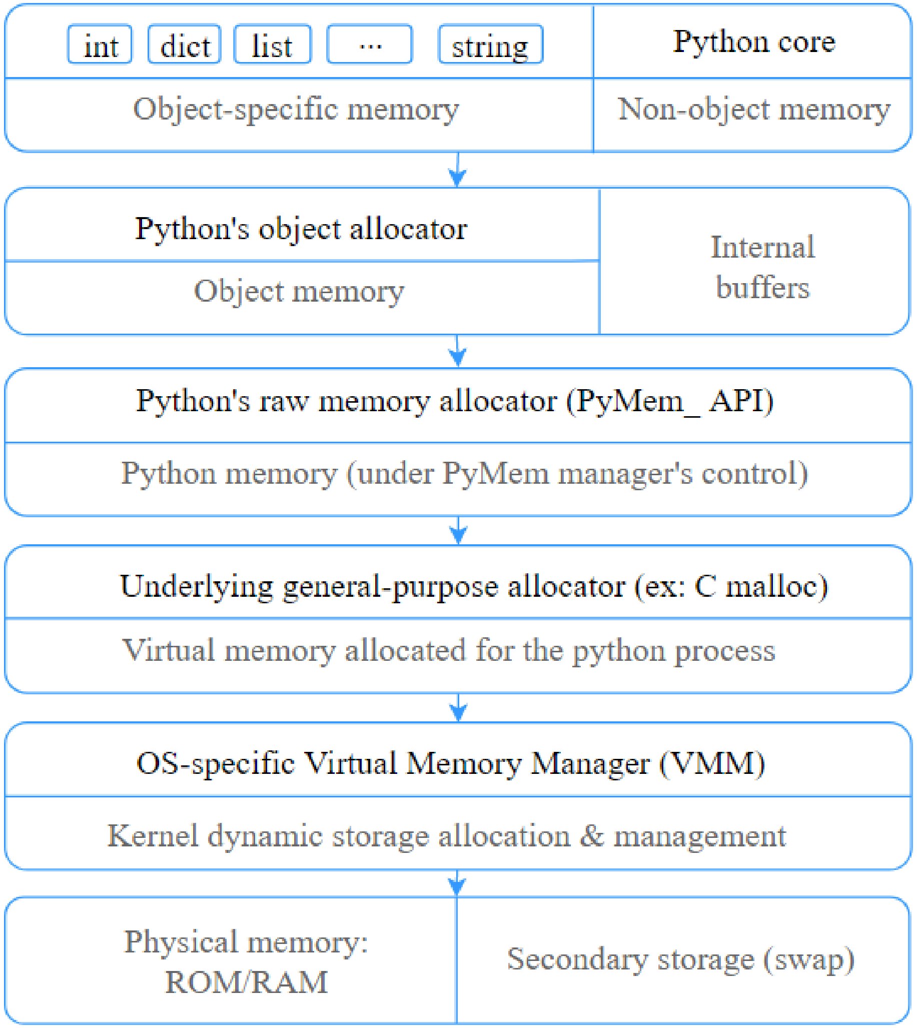
Memory Management

- For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs.
- The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

Memory Management

- It is important to understand that the management of the Python heap is performed by the interpreter itself.
- User has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap.
- The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

Memory Management



int | dict | list | ··· | string — Python core

Object-specific memory — Non-object memory

Python's object allocator — Internal buffers

Object memory

Python's raw memory allocator (PyMem_ API)

Python memory (under PyMem manager's control)

Underlying general-purpose allocator (ex: C malloc)

Virtual memory allocated for the python process

OS-specific Virtual Memory Manager (VMM)

Kernel dynamic storage allocation & management

Physical memory: ROM/RAM — Secondary storage (swap)

Memory Management

- Small object allocation
- To reduce overhead for small objects (less than 512 bytes) Python sub-allocates big blocks of memory.
- Larger objects are routed to standard C allocator. Small object allocator uses three levels of abstraction — arena, pool, and block.
- **Block**
- Block is a chunk of memory of a certain size. Each block can keep only one Python object of a fixed size. The size of the block can vary from 8 to 512 bytes and be a multiple of eight (i.e., 8-byte alignment). For convenience, such blocks are grouped in 64 size classes.

Memory Management

Request in bytes	Size of allocated block	size class idx
1-8	8	0
9-16	16	1
17-24	24	2
25-32	32	3
33-40	40	4
41-48	48	5
...	...	...
505-512	512	63

Memory Management

- **Pool**
- A collection of blocks of the same size is called a pool.
- Normally, the size of the pool is equal to the size of a memory page, i.e., 4Kb. Limiting pool to the fixed size of blocks helps with fragmentation.
- If an object gets destroyed, the memory manager can fill this space with a new object of the same size.

Memory Management

- **Each pool has three states:**
- **used — partially used, neither empty nor full**
- **full — all the pool's blocks are currently allocated**
- **empty — all the pool's blocks are currently available for allocation**
- **Arena**
- The arena is a chunk of 256kB memory allocated on the heap, which provides memory for 64 pools.

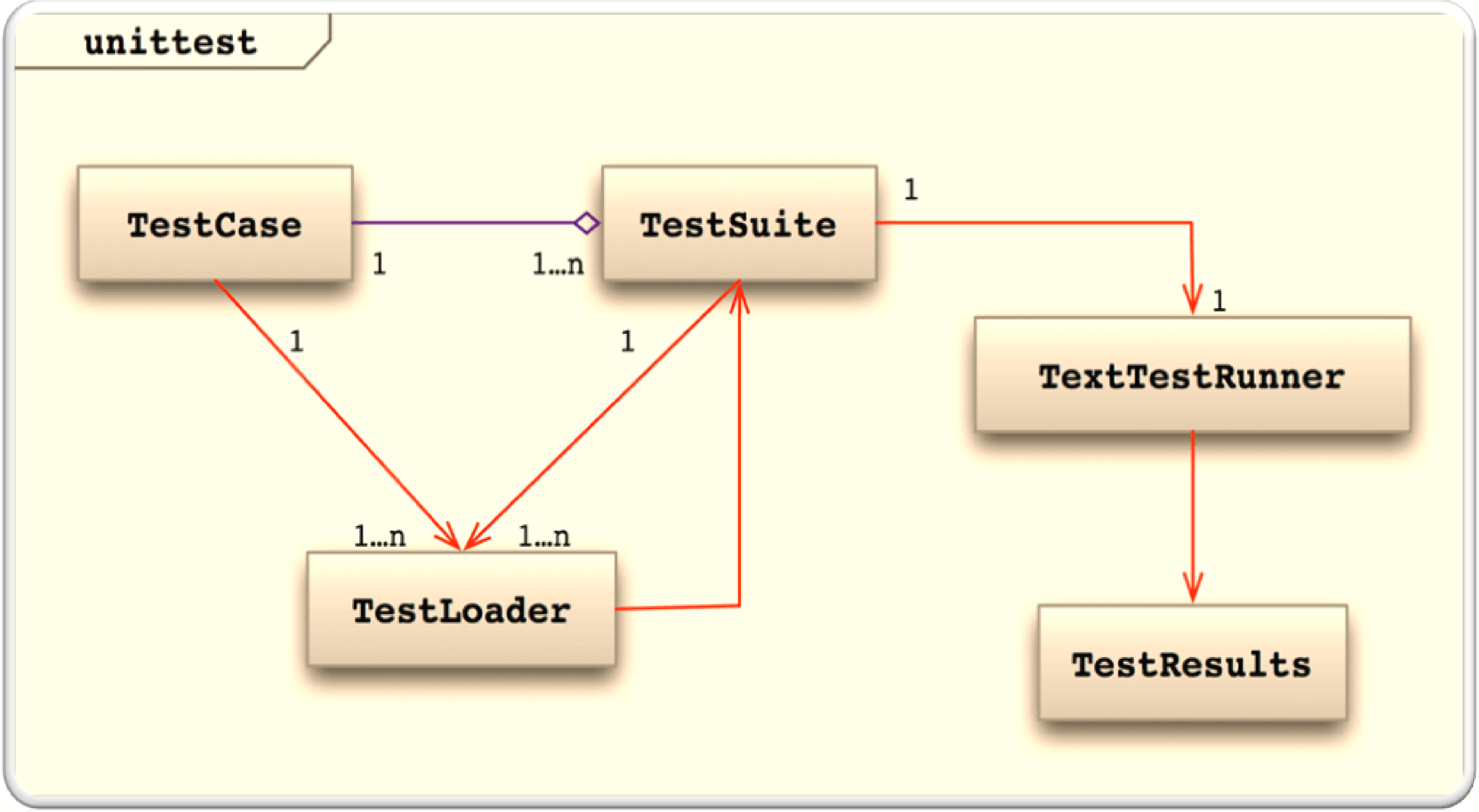
Memory Management

- **Memory deallocation**
- **Python's small object manager rarely returns memory back to the Operating System.**
- **An arena gets fully released If and only if all the pools in it are empty. For example, it can happen when you use a lot of temporary objects in a short period of time.**
- **Speaking of long-running Python processes, they may hold a lot of unused memory because of this behavior.**

Unit Testing in Python – Unittest

- **What is Unit Testing?**
  Unit Testing is the first level of software testing where the smallest testable parts of a software are tested. This is used to validate that each unit of the software performs as designed.
  The unittest test framework is python’s xUnit style framework.

Unit Testing in Python – Unittest



Unit Testing in Python – Unittest

- TestCase class: The TestCase class bears the test routines and delivers hooks for making each routine and cleaning up thereafter
- TestSuite class: It caters as a collection container, and it can possess multiple testcase objects and multiple testsuites objects
- TestLoader class: This class loads test cases and suites defined locally or from an external file. It emits a testsuite objects that posseses those suites and cases
- TextTestRunner class: To run the tests it caters a standard platform to execute the tests
- The TestResults class: It offers a standard container for the test results

Questions

Module Summary

- Regular Expressions
- Generators, Decorators and Iterators
- Socket and Network Programming
- Image Handling
- Processes and Threads
- Design Patterns