
PyPattyrn

```
1 from pypattyrn.creationl.singleton import Singleton
2
3 class DummyClass(object, metaclass=Singleton): # DummyClass is now a
    Singleton!
4     ...
```

PyPattyrn is a python package aiming to make it easier and faster to implement design patterns into your own projects.

Design patterns by nature cannot be directly translated into code as they are just a description of how to solve a particular problem. However, many of the common design patterns have boilerplate code that is common throughout all implementations of the pattern. This package captures that common code and makes it easy to use so that you dont have to write it yourself in all your projects. ____

Contents

- Installation
- Examples
 - Behavioral Patterns
 - * Chain of Responsibility
 - * Command
 - * Iterator
 - * Mediator
 - * Memento
 - * Null Object
 - * Observer
 - * Visitor
 - Creational Patterns
 - * Builder
 - * Factory
 - * Abstract Factory
 - * Object Pool
 - * Prototype
 - * Singleton
 - Structural Patterns

-
- * Adapter
 - * Composite
 - * Decorator
 - * Flyweight
- Resources
-

Installation

```
1 pip install pypattyrn
```

or

```
1 git clone https://github.com/tylerlaberge/PyPattyrn.git
2 cd PyPattyrn
3 python setup.py install
```

Examples

Behavioral Patterns Patterns which deal with communication between objects. ____ ##### Chain of Responsibility Pattern

Pass a request along a chain of objects until the request is handled.

```
1 from pypattyrn.behavioral.chain import Chain, ChainLink
2
3
4 class ConcreteChainLinkThree(ChainLink): # This object is a ChainLink
5
6     def handle(self, request): # Implement the handle method.
7         if request == 'handle_three':
8             return "Handled in chain link three"
9         else:
10            return self.successor_handle(request) # If this ChainLink
            can't handle the request,
```

```

11                                     # ask its successor
12                                     # to handle it.
13                                     # (Has no successor
                                     # so will raise
                                     # AttributeError)
                                     # (This exception is
                                     # caught and will
                                     # call a Chains fail
                                     # method)

14
15
16 class ConcreteChainLinkTwo(ChainLink): # This object is a ChainLink
17
18     def __init__(self): # Override init to set a successor on
        initialization.
19         super().__init__() # first call ChainLinks init
20         self.set_successor(ConcreteChainLinkThree()) # Set the
        successor of this chain link
21
        # to a
        ConcreteChainLinkThree
        instance.
22
23     def handle(self, request): # Implement the handle method.
24         if request == 'handle_two':
25             return "Handled in chain link two"
26         else:
27             return self.successor_handle(request) # If this ChainLink
        can't handle a request
28
        # ask its successor
        # to handle it
29
        # (the
        ConcreteChainLinkThree
        instance).
30
31
32 class ConcreteChainLinkOne(ChainLink): # This object is a ChainLink
33
34     def __init__(self):
35         super().__init__()
36         self.set_successor(ConcreteChainLinkTwo()) # Set the successor
        of this ChainLink
37
        # to a
        ConcreteChainLinkTwo
        instance.
38
39     def handle(self, request): # Implement the handle method.
40         if request == 'handle_one':
41             return "Handled in chain link one"
42         else:
43             return self.successor_handle(request) # If this ChainLink
        can't handle a request

```

```

44                                     # ask its successor
45                                     to handle it
46                                     # (the
47                                     ConcreteChainLinkTwo
48                                     instance).
49
50 class ConcreteChain(Chain): # This object is a Chain
51
52     def __init__(self): # Override init to initialize a Chain with the
53         starting chain link.
54         super().__init__(ConcreteChainLinkOne()) # Initialize this
55         Chain with a start chain link.
56
57                                     # (a
58                                     ConcreteChainLinkOne
59                                     instance)
60
61     def fail(self): # Implement the fail method, this is called if no
62         chain links could handle a request.
63         return 'Fail'
64
65 chain = ConcreteChain()
66
67 assert "Handled in chain link one" == chain.handle("handle_one")
68 assert "Handled in chain link two" == chain.handle("handle_two")
69 assert "Handled in chain link three" == chain.handle("handle_three")
70 assert "Fail" == chain.handle('handle_four')

```

Command Pattern Encapsulate information for performing an action into an object.

```

1 from pypattern.behavioral.command import Receiver, Command, Invoker
2
3
4 class Thermostat(Receiver): # This object is a Receiver.
5     # Contains methods for commands to use.
6
7     def raise_temp(self, amount):
8         return "Temperature raised by {0} degrees".format(amount)
9
10    def lower_temp(self, amount):
11        return "Temperature lowered by {0} degrees".format(amount)
12
13
14 class RaiseTempCommand(Command): # This object is a Command.
15
16    def __init__(self, receiver, amount=5): # Override init to include
17        a temperature amount argument.
18        super().__init__(receiver)
19        self.amount = amount

```

```

20     def execute(self): # Implement the execute method.
21         return self._receiver.action('raise_temp', self.amount) # Call
22             on the Receiver's action method with
23                                     # the
24                                     name
25                                     of
26                                     the
27                                     method
28                                     to
29                                     call
30                                     and
31                                     args
32                                     .
33                                     # (
34                                     Raises
35                                     the
36                                     temperature
37                                     by
38                                     5
39                                     degrees
40                                     .)
41
42     def unexecute(self): # Implement the unexecute method.
43         return self._receiver.action('lower_temp', self.amount) # Calls
44             on the Receiver to lower
45                                     # the
46                                     temperature
47                                     by
48                                     5
49                                     degrees
50                                     .
51
52     class LowerTempCommand(Command): # This object is a Command.
53
54         def __init__(self, receiver, amount=5):
55             super().__init__(receiver)
56             self.amount = amount
57
58         def execute(self):
59             return self._receiver.action('lower_temp', self.amount) # Call
60                 on the receiver to lower the
61                                     #
62                                     temperature
63                                     by
64                                     5
65                                     degrees
66                                     .
67
68
69

```

```

40     def unexecute(self):
41         return self._receiver.action('raise_temp', self.amount) # Call
                           on the receiver to raise the
42                                     #
                                     temperature
                                     by
                                     5
                                     degrees
                                     .
43
44
45     class Worker(Invoker): # This object is the Invoker
46
47         def __init__(self): # Override init to initialize an Invoker with
                               Commands to accept.
48             super().__init__([LowerTempCommand, RaiseTempCommand]) #
                               Initialize the Invoker with the
49                                     #
                                     LowerTempCommand
                                     and
                                     RaiseTempCommand
                                     classes
                                     .
50
51
52     thermostat = Thermostat() # Create a Receiver.
53     worker = Worker() # Create an Invoker.
54
55     assert "Temperature lowered by 5 degrees" == worker.execute(
        LowerTempCommand(thermostat)) # Have the Invoker execute a
        LowerTempCommand
56
57
58
57     assert "Temperature raised by 5 degrees" == worker.execute(
        RaiseTempCommand(thermostat)) # Have the Invoker execute a
        RaiseTempCommand
58

```

```
59 assert "Temperature lowered by 5 degrees" == worker.undo() # Undo the
    previous command (Calls the RaiseTempCommand unexecute method.)
```

Iterator Pattern A way of sequentially accessing elements in a collection.

```
1 from pypattern.behavioral.iterator import Iterable, Iterator
2
3
4 class Counter(Iterable): # This object is an Iterable
5
6     def __init__(self, max):
7         self.count = 0
8         self.max = max
9
10    def __next__(self): # Implement the __next__ method.
11        self.count += 1 # Increment the count
12        if self.count > self.max:
13            raise StopIteration() # make sure to raise StopIteration at
14                                   the appropriate time.
15        else:
16            return self.count - 1 # Return the count
17
18 class CounterIterator(Iterator): # This object is an Iterator
19
20    def __init__(self): # Override init to initialize an Iterator with
21        the Counter object.
22        super().__init__(Counter(10)) # Initialize the iterator with a
23        Counter that goes to 10.
24
25 counter_iterator = CounterIterator() # Create a CounterIterator.
26
27 for count in counter_iterator: # You can loop through it how you would
28     expect.
29     print(count) # 0, 1, 2, 3, ..., 9
```

Mediator Pattern An intermediary for managing communications between many objects.

```
1 from pypattern.behavioral.mediator import Mediator
2
3
4 class Dog(object):
5     sound = ''
6
7     def set_sound(self, sound):
8         self.sound = sound
9
10
11 class Cat(object):
12     sound = ''
13
14     def set_sound(self, sound):
15         self.sound = sound
16
17
18 dog = Dog()
19 cat = Cat()
20
21 mediator = Mediator() # Create a Mediator object.
22
23 mediator.connect('set_dog_sound', dog.set_sound) # Connect the signal '
24 set_dog_sound' to the dog.set_sound method.
25
26 mediator.connect('set_cat_sound', cat.set_sound) # Connect the signal '
27 set_cat_sound' to the cat.set_sound method.
28
29 mediator.connect('set_sound', dog.set_sound) # Also connect the signal
30 'set_sound' to the dog.set_sound method.
31
32 mediator.connect('set_sound', cat.set_sound) # Also connect the signal
33 'set_sound' to the cat.set_sound method.
34
35 mediator.signal('set_sound', 'foo') # Send out the signal 'set_sound'
36                                     # with an argument to be passed to
37                                     # the connected methods.
38                                     # (Sets dog.sound and cat.sound to
39                                     # 'foo')
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

```

44
45 mediator.disconnect('set_sound', dog.set_sound) # Disconnect the method
    dog.sound from the signal 'set_sound'
46
47 mediator.signal('set_sound', 'bar') # Send out the signal 'set_sound'
48                                     # (Only sets cat.sound to 'bar'
                                         because we disconnected dog.
                                         sound)
49
50 assert 'woof' == dog.sound
51 assert 'bar' == cat.sound

```

Memento Pattern Save the state of an object and rollback to it at a later time.

```

1 from pypattern.behavioral.memento import Originator
2
3
4 class Cat(Originator): # This object is an Originator
5     def __init__(self, name):
6         self.name = name
7
8
9 cat = Cat('Tom') # Initialize a Cat with the name 'Tom'
10 assert cat.name == 'Tom'
11
12 cat_memento = cat.commit() # Save the cats current state to a Memento
    object.
13 cat.name = 'jerry' # Change the cats name to 'jerry'
14 assert cat.name == 'jerry'
15
16 cat.rollback(cat_memento) # Restore the cats state to the memento
    object we saved.
17 assert 'Tom' == cat.name # The cats name was changed back to 'Tom' as
    expected.

```

Null Object Pattern Encapsulate the absence of an object into an object with neutral behavior.

```

1 from pypattern.behavioral.null import Null
2
3 # You can initialize a Null object with any parameters.
4 try:
5     Null()
6     Null('value')
7     Null('value', param='value')
8 except:
9     raise AssertionError()
10
11 null = Null()

```

```

12
13 # Attempting to set parameters won't do anything and won't error
14 try:
15     null.foo = 'foo'
16     null.foo.bar = 'bar'
17 except:
18     raise AssertionError()
19
20 # Calling on a null object in any way will just return that null object
21 .
22
23 assert null == null()
24 assert null == null('value')
25 assert null == null('value', param='value')
26 assert null == null.foo
27 assert null == null.foo.bar
28 assert null == null.foo.bar.method1()
29 assert null == null.foo.bar.method1().method2('foo', bar='bar').attr3
30
31 assert str(null) == '' # Null objects string representation is the
    empty string.
32 assert repr(null) == '' # Null objects repr is the empty string.
33 assert bool(null) is False # Null object evaluates to False as a
    boolean.
34
35 # Trying to delete attributes doesn't do anything and won't error.
36 try:
37     del null.foo
38     del null.foo.bar
39 except:
40     raise AssertionError()

```

Observer Pattern Notify many objects when a single object's state changes.

```

1 from pypattern.behavioral.observer import Observable, Observer
2
3
4 class ConcreteObservable(Observable): # This object is an Observable.
5     _kinda_private_var = 'I am kinda private'
6     __private_var = 'I am more private'
7
8     def change_state(self, **kwargs): # Some method to change this
        objects state and notify observers.
9         for key, value in kwargs.items():
10             setattr(self, key, value) # Change the objects state.
11
12         self.notify() # Notify all observers of the change in state.
13                        # Will call the update method of each attached
14                        # observer with kwargs that come
15                        # from this objects __dict__ attribute, minus any

```

```

15         private variables (starts with __ or _)
16
17     class ConcreteObserver(Observer): # This object is an Observer.
18         updated_state = None
19
20         def update(self, **state): # Implement the update method.
21             # Called when an attached observable
22             # calls its notify method.
23             self.updated_state = state
24
25     observable = ConcreteObservable() # Create an Observable
26     observer_1 = ConcreteObserver() # Create Observers
27     observer_2 = ConcreteObserver()
28     observer_3 = ConcreteObserver()
29
30     observable.attach(observer_1) # Attach each of the Observers to the
31     # Observable
32     observable.attach(observer_2)
33     observable.attach(observer_3)
34     observable.change_state(foo='foo', bar='bar') # Change the Observable's
35     # state.
36     # Also calls notify which
37     # will call each
38     # Observers update
39     # method.
40
41     expected_state = {'foo': 'foo', 'bar': 'bar'}
42
43     # Make sure each Observers state was changed accordingly when the
44     # notify method was called by the Observable.
45
46     assert sorted(expected_state.keys()) == sorted(observer_1.updated_state
47     .keys()) and \
48         sorted(expected_state.values()) == sorted(observer_1.
49     updated_state.values())
50
51     assert sorted(expected_state.keys()) == sorted(observer_2.updated_state
52     .keys()) and \
53         sorted(expected_state.values()) == sorted(observer_2.
54     updated_state.values())
55
56     assert sorted(expected_state.keys()) == sorted(observer_3.updated_state
57     .keys()) and \
58         sorted(expected_state.values()) == sorted(observer_3.
59     updated_state.values())
60
61     observable.detach(observer_1) # Detach Observer 1 from the Observable
62     observable.change_state(bar='foobar') # Change the Observables state.

```

```

52 expected_state_2 = {'foo': 'foo', 'bar': 'foobar'}
53
54 # Make sure each Observers state was changed accordingly when notify
   # was called by the Observable,
55 # Except for observer_1 because we detached that Observer from the
   # Observable.
56
57 assert sorted(expected_state_2.keys()) != sorted(observer_1.
   updated_state.keys()) or \
58     sorted(expected_state_2.values()) != sorted(observer_1.
   updated_state.values())
59
60 assert sorted(expected_state_2.keys()) == sorted(observer_2.
   updated_state.keys()) and \
61     sorted(expected_state_2.values()) == sorted(observer_2.
   updated_state.values())
62
63 assert sorted(expected_state_2.keys()) == sorted(observer_3.
   updated_state.keys()) and \
64     sorted(expected_state_2.values()) == sorted(observer_3.
   updated_state.values())

```

Visitor Pattern Add new operations to an existing class without modifying it.

```

1  from pypattern.behavioral.visitor import Visitee, Visitor
2
3
4  class A(Visitee): # This object is a Visitee
5      pass
6
7
8  class B(A): # This object is an A, which also makes this a Visitee.
9      pass
10
11
12 class C(B): # This object is a B, which also makes this a Visitee.
13     pass
14
15
16 class D(Visitee): # This object is a Visitee
17     pass
18
19
20 class NodeVisitor(Visitor): # This object is a Visitor
21
22     def generic_visit(self, node, *args, **kwargs): # Implement the
   generic visit method.
23
   # This is called
   # when there is no
   # visit_ method

```

```

24                                     # implemented for a
                                     particular
                                     visitee.
25
26     return 'generic_visit ' + node.__class__.__name__
27
28     def visit_b(self, node, *args, **kwargs): # Called when this object
        visits a Visitee of class 'B'.
29         return 'visit_b ' + node.__class__.__name__
30
31     def visit_d(self, node, *args, **kwargs): # Called when this object
        visits a Visitee of class 'D'.
32         return 'visit_d {0} args: {1} kwargs: {2}'.format(node.
            __class__.__name__, args, kwargs)
33
34 # Create Visitee's
35 node_a = A()
36 node_b = B()
37 node_c = C()
38 node_d = D()
39
40 # Create Visitor
41 node_visitor = NodeVisitor()
42
43 assert 'generic_visit A' == node_a.accept(node_visitor) # Visit node_a
        with the Visitor.
44                                     # Since the
                                     visitor does
                                     not have a
                                     visit_a
                                     method,
45                                     # generic_visit
                                     is called.
46 assert 'visit_b B' == node_b.accept(node_visitor) # Visit node_b with
        the Visitor.
47                                     # Calls the visitors
                                     visit_b method.
48 assert 'visit_b C' == node_c.accept(node_visitor) # Visit node_c with
        the Visitor.
49                                     # Even though the
                                     visitor does not
                                     have a visit_c
                                     method,
50                                     # since node_c
                                     inherits B, the
                                     Visitors visit_b
                                     method is called.
51
52 # Visit node_d with the Visitor. Calls the visitors visit_d method.
53 assert "visit_d D args: ('foo', 'bar') kwargs: {'foobar': 'foobar'}" ==
        node_d.accept(node_visitor,

```

54

55

Creational Patterns Patterns which deal with object creation. ____

Builder Pattern Separate object construction from its representation.

```
1 from pypattern.creational.builder import Builder, Director
2
3
4 class Building(object): # The object being constructed.
5
6     def __init__(self):
7         self.floor = None
8         self.size = None
9
10    def __repr__(self):
11        return 'Floor: {0.floor} | Size: {0.size}'.format(self)
12
13
14 class HomeBuilder(Builder): # A base Builder class for constructing
    homes.
15
16    def __init__(self):
17        super().__init__(Building()) # Initialize the Builder class
            with a Building instance.
18        self._register('floor', self._build_floor) # Register the
            keyword 'floor' with the _build_floor method.
19        self._register('size', self._build_size) # Register the
            keyword 'size' with the _build_size method.
20
```

```

21     def _build_floor(self):
22         pass
23
24     def _build_size(self):
25         pass
26
27
28 class HouseBuilder(HomeBuilder): # A concrete HomeBuilder class for
    constructing houses
29
30     def _build_floor(self):
31         self.constructed_object.floor = 'One' # Alter the Building's
            floor attribute.
32
33     def _build_size(self):
34         self.constructed_object.size = 'Big' # Alter the Building's
            size attribute.
35
36
37 class FlatBuilder(HomeBuilder): # A concrete HomeBuilder class for
    constructing flats
38
39     def _build_floor(self):
40         self.constructed_object.floor = 'More than one' # Alter the
            Building's floor attribute.
41
42     def _build_size(self):
43         self.constructed_object.size = 'Small' # Alter the Building's
            size attribute.
44
45
46 class HomeDirector(Director): # A Director class for managing home
    construction.
47
48     def construct(self):
49         self.builder.build('floor') # Build the floor part of the
            Building by using the keyword 'floor'
50         self.builder.build('size') # Build the size part of the
            Building by using the keyword 'size'
51
52
53 home_director = HomeDirector()
54
55 home_director.builder = HouseBuilder() # Use the house builder.
56 home_director.construct() # Construct the house.
57 house = home_director.get_constructed_object() # Get the constructed
    house.
58 print(repr(house)) #Floor: One | Size: Big
59
60 home_director.builder = FlatBuilder() # Use the flat builder.
61 home_director.construct() # Construct the flat.

```

```
62 house = home_director.get_constructed_object() # Get the constructed
    flat.
63 print(repr(house))      #Floor: More than one | Size: Small
```

Factory Pattern An interface for creating an object.

```
1 from pypattern.creational.factory import Factory # This is just an
    interface
2
3
4 class Cat(object):
5
6     def speak(self):
7         print('meow')
8
9
10 class Dog(object):
11
12     def speak(self):
13         print('woof')
14
15
16 class AnimalFactory(Factory): # A factory class for creating animals.
17
18     def create(self, animal_type): # Implement the abstract create
        method.
19         if animal_type == 'cat':
20             return Cat()
21         elif animal_type == 'dog':
22             return Dog()
23         else:
24             return None
25
26
27 animal_factory = AnimalFactory()
28
29 cat = animal_factory.create('cat')
30 dog = animal_factory.create('dog')
31
32 cat.speak() # 'meow'
33 dog.speak() # 'woof'
```

Abstract Factory Pattern Create an instance from a family of factories.

```
1 from pypattern.creational.factory import Factory, AbstractFactory
2
3
4 class Cat(object):
```

```
5
6     def speak(self):
7         print('meow')
8
9
10    class Dog(object):
11
12        def speak(self):
13            print('woof')
14
15
16    class Ant(object):
17
18        def march(self):
19            print('march')
20
21
22    class Fly(object):
23
24        def fly(self):
25            print('fly')
26
27
28    class AnimalFactory(Factory): # A factory class for creating animals.
29
30        def create(self, animal_type): # Implement the abstract create
31            method.
32            if animal_type == 'cat':
33                return Cat()
34            elif animal_type == 'dog':
35                return Dog()
36            else:
37                return None
38
39    class InsectFactory(Factory): # A factory class for creating insects.
40
41        def create(self, insect_type): # Implement the abstract create
42            method.
43            if insect_type == 'ant':
44                return Ant()
45            elif insect_type == 'fly':
46                return Fly()
47            else:
48                return None
49
50    class CreatureFactory(AbstractFactory): # A Factory class for creating
51        creatures.
52
53        def __init__(self):
```

```

53     super().__init__()
54     self._register('insect_factory', InsectFactory()) # Register
        an InsectFactory with a keyword.
55     self._register('animal_factory', AnimalFactory()) # Register
        an AnimalFactory with a keyword.
56
57     def create(self, creature_type): # Implement the Abstract create
        method.
58         if creature_type == 'cat' or creature_type == 'dog':
59             return self._factories['animal_factory'].create(
                creature_type) # Use the AnimalFactory
60         elif creature_type == 'ant' or creature_type == 'fly':
61             return self._factories['insect_factory'].create(
                creature_type) # Use the InsectFactory
62         else:
63             return None
64
65     creature_factory = CreatureFactory()
66
67     cat = creature_factory.create('cat')
68     dog = creature_factory.create('dog')
69     ant = creature_factory.create('ant')
70     fly = creature_factory.create('fly')
71
72     cat.speak() # 'meow'
73     dog.speak() # 'woof'
74     ant.march() # 'march'
75     fly.fly() # 'fly'

```

Object Pool Pattern Provide a pool of instantiated objects which can be checked out and returned rather than creating new objects all the time.

```

1  from pypattern.creational.pool import Reusable, Pool
2
3
4  class Dog(Reusable):
5      def __init__(self, sound):
6          self.sound = sound
7          super().__init__()
8
9
10 class DogPool(Pool):
11     def __init__(self):
12         super().__init__(Dog, 'woof')
13
14
15 dog_pool = DogPool()
16
17 dog_one = dog_pool.acquire()

```

```
18 dog_two = dog_pool.acquire()
19 dog_two.sound = 'meow'
20
21 dog_pool.release(dog_one)
22 dog_three = dog_pool.acquire()
23
24 dog_pool.release(dog_two)
25 dog_four = dog_pool.acquire()
26
27 assert id(dog_one) == id(dog_three)
28 assert id(dog_two) == id(dog_four)
29 assert dog_one.sound == dog_two.sound
30 assert dog_three.sound == dog_four.sound
31 assert dog_one.sound == dog_four.sound
```

Prototype Pattern Clone an object to produce new objects.

```
1 from pypattern.creational.prototype import Prototype
2
3
4 class Point(Prototype):
5     def __init__(self, x, y):
6         self.x = x
7         self.y = y
8
9     def move(self, x, y):
10        self.x += x
11        self.y += y
12
13
14 point_one = Point(15, 15)
15 point_two = point_one.prototype(z=20)
16 point_three = point_two.prototype()
17
18 assert point_one.x == point_two.x
19 assert point_one.y == point_two.y
20 assert not hasattr(point_one, 'z')
21 assert hasattr(point_two, 'z')
22 assert point_two.z == 20
23 assert point_three.__dict__ == point_two.__dict__
24
25 from math import sqrt
26 def distance_to(this, other):
27     return sqrt((this.x - other.x) ** 2 + (this.y - other.y) ** 2)
28
29 point_four = point_three.prototype(distance_to=distance_to)
30
31 assert hasattr(point_four, 'distance_to')
32 assert point_four.distance_to(point_three) == 0
```

Singleton Pattern Ensure that only a single instance of a class exists.

```
1 from pypattern.creational.singleton import Singleton
2
3
4 class DummySingletonOne(object, metaclass=Singleton):
5
6     def __init__(self):
7         pass
8
9
10 class DummySingletonTwo(object, metaclass=Singleton):
11
12     def __init__(self):
13         pass
14
15
16 dummy_class_one_instance_one = DummySingletonOne()
17 dummy_class_one_instance_two = DummySingletonOne()
18
19 dummy_class_two_instance_one = DummySingletonTwo()
20 dummy_class_two_instance_two = DummySingletonTwo()
21
22 assert id(dummy_class_one_instance_one) == id(
23     dummy_class_one_instance_two)
24
25 assert id(dummy_class_two_instance_one) == id(
26     dummy_class_two_instance_two)
```

Structural Patterns Patterns which deal with object composition ____

Adapter Pattern Wrap an object into an interface which the client expects.

```
1 from pypattern.structural.adapter import Adapter
2
3
4 class Dog(object):
5     def __init__(self):
6         self.name = "Dog"
7
8     def bark(self):
9         return "woof!"
```

```
10
11
12 class Cat(object):
13     def __init__(self):
14         self.name = "Cat"
15
16     def meow(self):
17         return "meow!"
18
19 cat = Cat()
20 dog = Dog()
21
22 cat_adapter = Adapter(cat, make_noise=cat.meow)
23 dog_adapter = Adapter(dog, make_noise=dog.bark)
24
25 assert cat_adapter.make_noise == cat.meow
26 assert dog_adapter.make_noise == dog.bark
27
28 assert cat_adapter.make_noise() == 'meow!'
29 assert dog_adapter.make_noise() == 'woof!'
30
31 assert cat_adapter.name == 'Cat'
32 assert dog_adapter.name == 'Dog'
33
34 assert cat_adapter.meow() == 'meow!'
35 assert dog_adapter.bark() == 'woof!'
36
37 assert cat_adapter.original_dict() == cat.__dict__
38 assert dog_adapter.original_dict() == dog.__dict__
39
40 bad_cat_adapter = Adapter(cat, foo=cat.name)
41
42 try:
43     bad_cat_adapter.foo
44 except AttributeError:
45     pass
46 else:
47     raise AssertionError()
48
49 bad_dog_adapter = Adapter(dog, make_noise=cat.meow)
50
51 try:
52     bad_dog_adapter.make_noise()
53 except AttributeError:
54     pass
55 else:
56     raise AssertionError()
```

Composite Pattern Compose objects into a tree structure of objects that can be treated uniformly.

```
1 from pypattn.structural.composite import Composite
2
3
4 class Component(object):
5
6     def do_something(self):
7         pass
8
9
10 class Leaf(Component):
11
12     def __init__(self):
13         self.did_something = False
14
15     def do_something(self):
16         self.did_something = True
17
18
19 leaf_one = Leaf()
20 leaf_two = Leaf()
21 leaf_three = Leaf()
22
23 composite_one = Composite(Component)
24 composite_two = Composite(Component)
25 composite_three = Composite(Component)
26
27 composite_one.add_component(leaf_one)
28 composite_two.add_component(leaf_two)
29 composite_three.add_component(leaf_three)
30
31 composite_two.add_component(composite_three)
32 composite_one.add_component(composite_two)
33
34
35 assert set() == {leaf_one, composite_two}.symmetric_difference(
36     composite_one.components)
37 assert set() == {leaf_two, composite_three}.symmetric_difference(
38     composite_two.components)
39 assert set() == {leaf_three}.symmetric_difference(composite_three.
40     components)
41
42 composite_two.remove_component(composite_three)
43
44 assert set() == {leaf_two}.symmetric_difference(composite_two.
45     components)
46
47 assert not leaf_one.did_something
48 assert not leaf_two.did_something
```

```
45 assert not leaf_three.did_something
46
47 composite_one.do_something()
48
49 assert leaf_one.did_something
50 assert leaf_two.did_something
51 assert not leaf_three.did_something
```

Decorator Pattern Attach additional functionality to functions.

```
1 import time
2 from pypattern.structural.decorator import DecoratorSimple,
   DecoratorComplex, CallWrapper
3
4
5 class TimeThis(DecoratorSimple):
6
7     def __call__(self, *args, **kwargs):
8         start = time.time()
9         result = self.func(*args, **kwargs)
10        end = time.time() - start
11        return result, end
12
13
14 class SlowClass(object):
15
16     @TimeThis
17     def slow_function(self, n):
18         time.sleep(n)
19         return 'foo'
20
21 slow_class = SlowClass()
22 result = slow_class.slow_function(1)
23 assert result[0] == 'foo'
24 assert 1 <= result[1] <= 2
25
26
27 class Alert(DecoratorComplex):
28
29     def __init__(self, alert_time):
30         self.alert_time = alert_time
31
32     @CallWrapper
33     def __call__(self, func, *args, **kwargs):
34         start = time.time()
35         return_val = func(*args, **kwargs)
36         end = time.time() - start
37         if end > self.alert_time:
38             return return_val, True
39         return return_val, False
```

```
40
41
42 class SlowClass(object):
43
44     @Alert(1)
45     def slow_function_true(self, n):
46         time.sleep(n)
47         return n
48
49     @Alert(1)
50     def slow_function_false(self, n):
51         return n
52
53
54 slow_class = SlowClass()
55 assert (2, True) == slow_class.slow_function_true(2)
56 assert (10, False) == slow_class.slow_function_false(10)
```

Flyweight Pattern Share data with other similar objects to increase efficiency.

```
1 from pypattern.structural.flyweight import FlyweightMeta
2
3
4 class Card(object, metaclass=FlyweightMeta):
5
6     def __init__(self, suit, value):
7         self.suit = suit
8         self.value = value
9
10
11 three_of_spades = Card('Spade', 3)
12 four_of_spades = Card('Spade', 4)
13 three_of_spades_two = Card('Spade', 3)
14
15 assert id(three_of_spades) == id(three_of_spades_two)
16 assert id(three_of_spades) != id(four_of_spades)
```

Resources

- [API Documentation](#)
 - [General Design Pattern Information](#)
-