See also https://en.wikipedia.org/wiki/Software_design_pattern

Creational Patterns

1. **Factory** centralises creation of object instances into a function(s), so all object instance creations can easily be found. Decouples generation of object from accessing objection.
2. **Abstract factory** extends factory method (which centralises object instance creation). It contains a collection of object instance factories which may be chosen from (e.g. forms styled for Windows or Mac depending on OS).
3. **Builder** constructs a complex object from component objects (often where order of construction is important) - e.g. build web page from component parts (objects). The *Director* controls the construction. Returns a single final object (built from component objects).
4. **Prototype** creates objects by *cloning* an existing object. Can also be useful for create an archive copy of an object at a given point in time. Python has built in clone method for this.
5. **Singleton** is a class which allows only one instance of the class (e.g. for storing the global state of a program, or for a coordinating object, or controlling accesses to a shared resource).

Structural Design Patterns

6. **Adapter** is a class of objects for creating an interface between two otherwise incompatible objects (e.g. interfacing between old and new software without needing to change either the old or new object architecture, or when using software where the source code is hidden). It is based on a dictionary that matches `new_object.method()` to `old_object_method()`.
7. **Decorator** extends the function of a class, object or method. Especially useful when many different functions require the same extension, e.g. timing function, validating inputs, monitoring/logging, adding GUI components.
8. **Bridge** provides an interface between general code and specific use cases, e.g. processing data with a bridge that allows multiple sources of data, device drivers allowing generic output targeted to specific devices, or using alternative themed GUI implementations.
9. **Facade** provides a simplified application interface to the client, hiding the underlying complex application. Client code may then call just one class/method. Alternatively, in a complex system, parts of the system

may communicate with each other through a limited number of facades. Facades can also help to keep client cod access independent of underlying code.

10. **Flyweight** minimises memory usage by sharing resources as much as possible. A flyweight object contains state-independent immutable data (*intrinsic* data) that is used by other objects. Flyweight objects should not contain state-dependent mutable data (*extrinsic* data). Look for what data is common across objects and extract that to a flyweight object.

11. **Model-View-Controller (MVC)** uses the principle of *Separation of Concerns (SoC)* where an application is divided into distinct sections. MVC describes application of SoC to OOP. 1) *Model*: the core of the program (data, logic, rules, state). 2) *View*: visual representation of the model, such as a GUI, keyboard or mouse input, text output, chart output, etc. 3) *Controller*: the link between the model and the view. MVC allows separation of view and model via different controllers, allowing for different views in different circumstances. MVC may also make use of different *adaptors*.

12. **Proxy** uses a *surrogate* object to access an actual object. Examples are 1) *remote proxy* which represents an object that is held elsewhere, 2) *virtual proxy* which uses lazy implementation to delay creation of an object until and if actually required, 3) *protection proxy* controls access to a sensitive object, and 4) *smart (reference) proxy* which performs extra actions as an object is accessed (e.g. counting number of times object accessed, or thread-safety checking).

Behavioural Design Patterns

13. **Chain of Responsibility** is used when it is not known which object will respond to a request. The request is sent to multiple objects (e.g. nodes on a network). AN object will then decide whether to accept the request, forward the request, or reject the request. This chain continues until all required actions are completed.The Chain of Responsibility creates a flow of requests to achieve a task. The flow of requests is worked out at the time, rather than the route being pre-defined. The client only needs to know how to communicate with the head of the chain.

14. **Command** encapsulates all required actions in a command, often including ability to undo. GUI buttons may also use command to execute optoin or display tool tips. Macros may be an example of command: a sequence of steps to perform.

15. **Observer** (or **Publish/Subscribe**) defines a one-to-many dependency between objects where a state change in one object results in all its

dependents being notified and updated automatically.

16. **State** allows an object to alter its behavior when its internal state changes. The object will appear to change its behaviour. For example in a game amonster might go from sleep to wake to attack. Python has the module state_machine to assist in state machine programming.
17. **Interpreter** is a simple langauge to allow advanced users of a system more control in the system.
18. **Strategy** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it. For example Python may select a specific sort method depending on the data.
19. **Memento** supports history and undo, and enables restore to a previous state.
20. **Iterator** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
21. **Template** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Microservices and patterns for cloud

22. **Microservices** breaks applications down into smaller applications, each of which may be developed and deployed separately. Frequently each microservice will run in a *container* that also has all the required dependencies.
23. **Retry** is common in micrososervices, and allows for temporary unavailability of a dependent services. The process may be retired, either immediately or after waiting a few seconds.
24. **Circuit Breaker** monitors a service and shuts down a service if reliability is too low.
25. **Cache-Aside** holds commonly used data in memory rather than re-reading from disc.
26. **Throttling** limits the number of requests a client can send.