**What is dependency injection?**

Dependency injection is a programming technique that makes a class independent of its dependencies. It achieves that by decoupling the usage of an object from its creation. This helps you to follow SOLID’s [dependency inversion](https://stackify.com/dependency-inversion-principle/) and [single responsibility principles](https://stackify.com/solid-design-principles/).

### **The 4 roles in dependency injection**

1. The **service** you want to use.
2. The **client** that uses the service.
3. An **interface** that’s used by the client and implemented by the service.
4. The **injector** which creates a service instance and injects it into the client.

Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

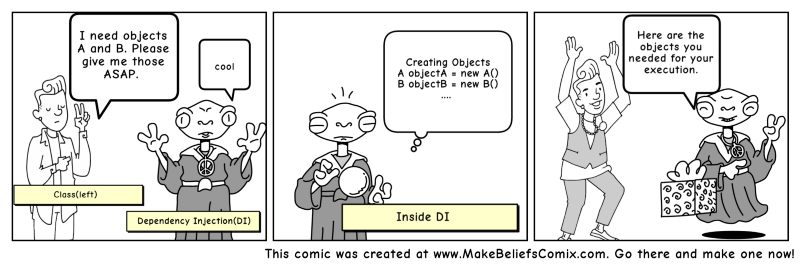
 DI pattern separates the responsibility of creating an object of the service class out of the client class.

## **Types of Dependency Injection**

**Constructor Injection**

**Property Injection**

**Method Injection**



***What are threads?***

Threads refer to the highest level of code executed by a processor, so with many threads, your CPU can handle several tasks at the same time. All CPUs have active threads, and every process performed on your computer has at least a single thread.

Number of threads depend on number of cores in CPU,each core have 2 threads.

Threads determine how many tasks can computer perform at any given time.

A thread is a small sequence of programmed instructions. Threads refer to the highest level of code your processor can execute.

They are usually managed by a scheduler, which is a standard part of any operating system.

 The more threads you have, the better your computer’s performance will be. With multiple threads, a single process can handle a variety of tasks simultaneously.

A thread is the unit of execution within a process. A process can have anywhere from just one thread to many threads.

Threads are sometimes called lightweight processes because they have their own stack but can access shared data. Because threads share the same address space as the process and other threads within the process, the operational cost of communication between the threads is low, which is an advantage. The disadvantage is that a problem with one thread in a process will certainly affect other threads and the viability of the process itself.

Context switching between threads of the same process is less expensive

• They have their own ID, PC, a register set, and stack

• But they share with other threads in the same process code, data, heap sections, and other OS resources, such as open files, permissions

Each thread belongs to exactly one process and no thread can exist outside a process.

♣ Thread switching is a type of context switching from one thread to another thread in the same process.

♣ It is very efficient and much cheaper because • It involves switching out only identities and resources such as the program counter, registers and stack pointers.

• While, processes switching involves switching of all the process resources. 4Such as, memory addresses, page tables, and kernel resources, caches in the processor

Threads are more easily affected by problems caused by other threads in the same process.

**Types of Threads**

♣ **User Level Threads** − management done by user.

♣ **Kernel Level Threads** − Operating System manages threads acting on kernel. • Supported by virtually all general - purpose operating systems, including: Windows, Linux, Mac OS X, iOS, Android

♣ User threads are simply called threads, and lightweight process refers to kernel threads.

**User Threads**

♣ Kernel is not aware of the existence of these threads.

It handles them as if they were singlethreaded processes.

• The application starts with a single thread.

• The thread library contains APIs for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts.

• There is no kernel involvement in synchronization for user-level threads.

♣ User-level threads are much faster than kernel level threads.

• There are no kernel mode privileges required for thread switching.

♣ But

• Cannot use multiprocessing to their advantage.

• The entire process is blocked if one user-level thread performs blocking operation.

**Kernel Threads**

♣ The application has no direct control over these threads • The Kernel performs thread creation, scheduling, switching and management in kernel space.

4requires a mode switch to the Kernel.

♣ Kernel threads are generally slower to create and manage than the user threads.

♣ Kernel threads are strongly implementationdependent. To facilitate the writing of portable programs, libraries provide user threads.

♣ Kernel routines (device drivers) themselves can be multithreaded.

♣ A kernel thread is the schedulable entity, which means scheduling by the Kernel is done on a thread basis.

• Kernel can simultaneously schedule multiple threads from the same process.

• If one thread in a process is blocked, the Kernel can schedule another thread of the same process.

• The context information for the process as well as the process threads is all managed by the kernel ⎝ generally slower .

**Why recursive is faster than iterative?**

The **recursive function runs much faster than the iterative one**. The reason is because in the latter, for each item, a CALL to the function st\_push is needed and then another to st\_pop . In the former, you only have the recursive CALL for each node. Plus, accessing variables on the callstack is incredibly fast.

*Memoization makes recursion palatable.*

**memoization is an optimization technique** that makes applications more efficient and hence faster. It does this by storing computation results in cache, and retrieving that same information from the cache the next time it's needed instead of computing it again.

In simpler words, it consists of storing in **cache** the output of a function, and making the function check if each required computation is in the cache before computing it.

A **cache** is simply a temporary data store that holds data so that future requests for that data can be served faster.

**What is clean code principles?**

## General rules

1. Follow standard conventions.
2. Keep it simple stupid. Simpler is always better. Reduce complexity as much as possible.
3. Boy scout rule. Leave the campground cleaner than you found it.
4. Always find root cause. Always look for the root cause of a problem.

## Understandability tips

1. Be consistent. If you do something a certain way, do all similar things in the same way.
2. Use explanatory variables.
3. Encapsulate boundary conditions. Boundary conditions are hard to keep track of. Put the processing for them in one place.
4. Prefer dedicated value objects to primitive type.
5. Avoid logical dependency. Don't write methods which works correctly depending on something else in the same class.
6. Avoid negative conditionals.

## Names rules

1. Choose descriptive and unambiguous names.
2. Make meaningful distinction.
3. Use pronounceable names.
4. Use searchable names.
5. Replace magic numbers with named constants.
6. Avoid encodings. Don't append prefixes or type information.

## Functions rules

1. Small.
2. Do one thing.
3. Use descriptive names.
4. Prefer fewer arguments.
5. Have no side effects.
6. Don't use flag arguments. Split method into several independent methods that can be called from the client without the flag.

## Comments rules

1. Always try to explain yourself in code.
2. Don't be redundant.
3. Don't add obvious noise.
4. Don't use closing brace comments.
5. Don't comment out code. Just remove.
6. Use as explanation of intent.
7. Use as clarification of code.
8. Use as warning of consequences.

## Source code structure

1. Separate concepts vertically.
2. Related code should appear vertically dense.
3. Declare variables close to their usage.
4. Dependent functions should be close.
5. Similar functions should be close.
6. Place functions in the downward direction.
7. Keep lines short.
8. Don't use horizontal alignment.
9. Use white space to associate related things and disassociate weakly related.
10. Don't break indentation.

## Objects and data structures

1. Hide internal structure.
2. Prefer data structures.
3. Avoid hybrids structures (half object and half data).
4. Should be small.
5. Do one thing.
6. Small number of instance variables.
7. Base class should know nothing about their derivatives.
8. Better to have many functions than to pass some code into a function to select a behavior.
9. Prefer non-static methods to static methods.

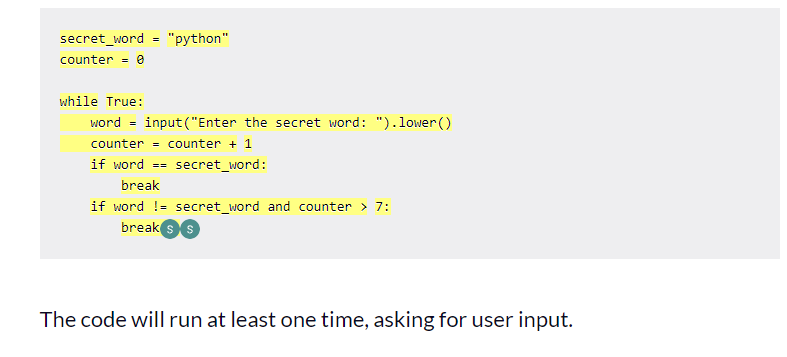
## Tests

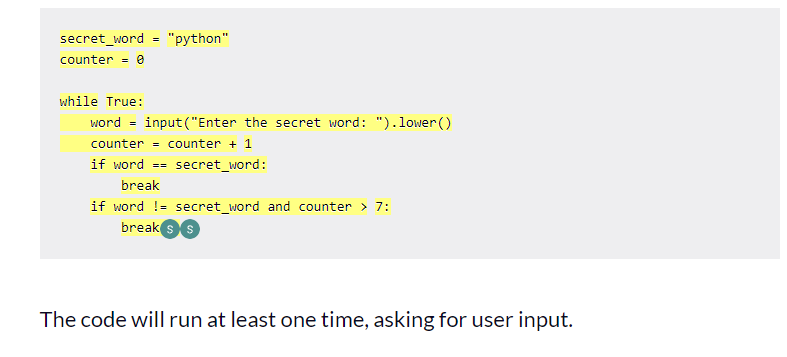
1. One assert per test.
2. Readable.
3. Fast.
4. Independent.
5. Repeatable.

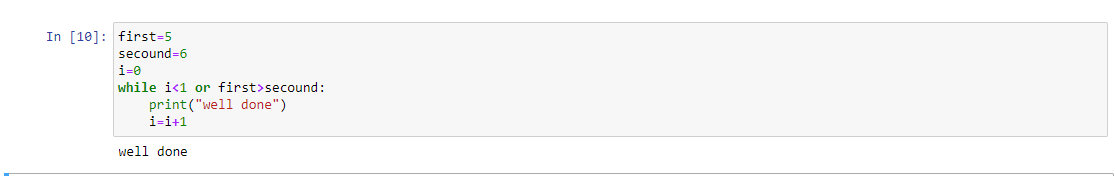
## Code smells

1. Rigidity. The software is difficult to change. A small change causes a cascade of subsequent changes.
2. Fragility. The software breaks in many places due to a single change.
3. Immobility. You cannot reuse parts of the code in other projects because of involved risks and high effort.
4. Needless Complexity.
5. Needless Repetition.
6. Opacity. The code is hard to understand.

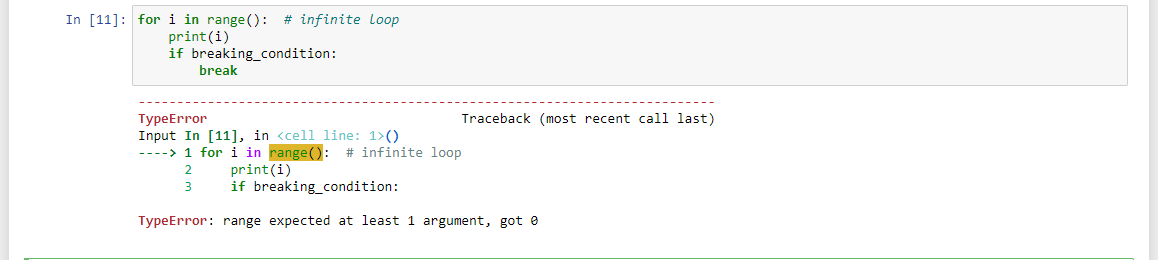
**To make do while in python**

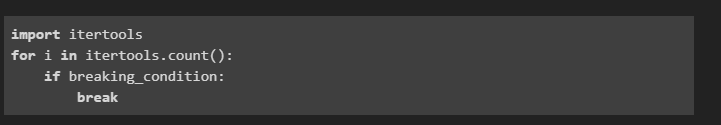
****

****

******

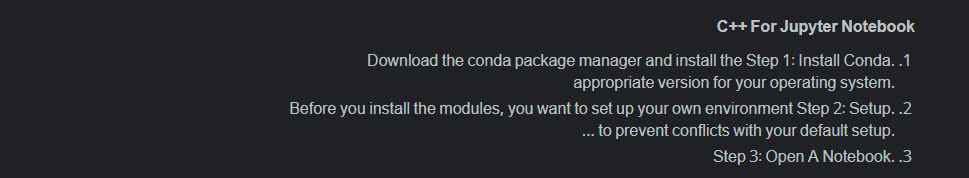
***To make infinite for loop in python***

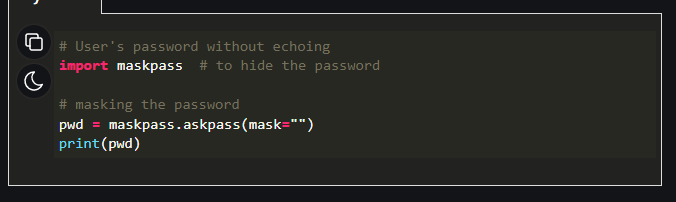
******

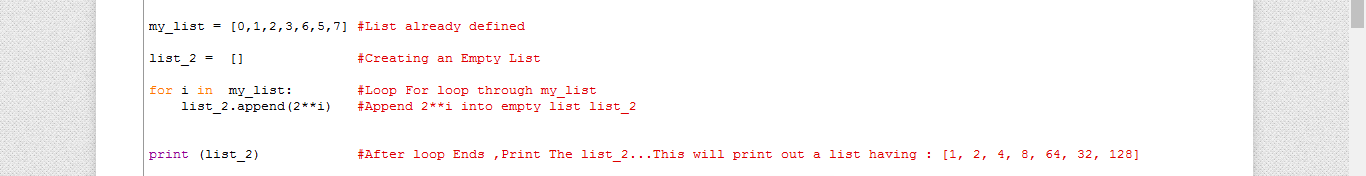
******

***Can we ran c++ in jupyter?***

Yes,by Xeus-Cling

To hide password while writing ?





Last task to make loop increamented by power2.