**OCAML**As with all modern languages, OCaml provides a garbage collector so that you don't need to explicitly allocate and free memory as in C/C++.  
The OCaml garbage collector is a modern hybrid generational/incremental collector which outperforms hand-allocation in most cases. Unlike the Java GC, which gives GCs a bad name, the OCaml GC doesn't allocate huge amounts of memory at start-up, nor does it appear to have arbitrary fixed limits that need to be overridden by hand.  
Why would garbage collection be faster than explicit memory allocation as in C? It's often assumed that calling free costs nothing. In fact free is anexpensive operation which involves navigating over the complex data structures used by the memory allocator. If your program calls free intermittently, then all of that code and data needs to be loaded into the cache, displacing your program code and data, each time you free a single memory allocation. A collection strategy which frees multiple memory areas in one go (such as either a pool allocator or a GC) pays this penalty only once for multiple allocations (thus the cost per allocation is much reduced).  
GCs also move memory areas around and compact the heap. This makes allocation easier, hence faster, and a smart GC can be written to interact well with the L1 and L2 caches.  
Of course none of this precludes writing a very fast hand-allocator, but it's considerably harder work than most programmers realise.  
OCaml's garbage collector has two heaps, the **minor heap** and the **major heap**. This recognises a general principle: Most objects are small and allocated frequently and then immediately freed. These objects go into the minor heap first, which is GCed frequently. Only some objects are long lasting. These objects get promoted from the minor heap to the major heap after some time, and the major heap is only collected infrequently.  
The OCaml GC is synchronous. It doesn't run in a separate thread, and it can only get called during an allocation request

🡺 **Type checking in Ocaml**

OCaml has structural typing for objects rather than nominative typing as in Java. So the type of an object is basically determined (and only determined) by its methods. Objects in OCaml can be created directly, without going through something like a class.  
You can write functions which require that its argument objects have certain methods (and that those methods have certain types); for example, the following method takes an argument that is any object with a method "bar":  
let foo x = x#bar

**JAVA**

**🡺 Namespace in Java Means?**

Namespaces are named program regions used to limit the scope of variables inside the program. They are used in many programming languages to create a separate region for a group of variables, functions, classes, etc. The usage of namespaces helps to avoid conflict with the existing definitions. Namespaces provide a way of implementing information hidden. Some examples of namespaces in Java are classes and packages.

**🡺 if one of the threads calls garbage collection, will the un-referenced objects in other threads be collected also?**

Yes or No

Yes. Objects are not "in threads" -- there is a single object graph for all threads running in the program, so when GC happens, unreachable objects are collected regardless of which thread created them, or had local references to them.

No. When a thread calls Runtime.gc() the VM is not obliged to actually do anything so it may be that no GC happens and no memory is collected. For example it has no effect when -XX:+DisableExplicitGC is specified at the command line.

[Runtime.gc()](http://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html#gc%28%29)

Calling this method **suggests** that the Java virtual machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse.

No. Even when a GC happens, not all unreachable objects that were only ever reachable from one thread's stack will necessarily be collected since generational GCs only deal with a subset of the object graph, and if that subset happens to contain all the unreachable objects created by a particular thread then it is only coincidence.

**🡺Java thread GC or not?**A running thread is considered a so called garbage collection root and is one of those things keeping stuff from being garbage collected. When the garbage collector determines whether your object is '[reachable](https://web.archive.org/web/20090831141004/http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html)' or not, it is always doing so using the set of garbage collector roots as reference points.Consider this, why is your main thread not being garbage collected, no one is referencing that one either.

### **🡺 Advantage and Disadvantage of GC** == Advantages Garbage collection frees the programmer from manually dealing with memory deallocation. As a result, certain categories of [bugs](https://en.wikipedia.org/wiki/Bug_(software)) are eliminated or substantially reduced:

* [*Dangling pointer*](https://en.wikipedia.org/wiki/Dangling_pointer)*bugs*, which occur when a piece of memory is freed while there are still [pointers](https://en.wikipedia.org/wiki/Pointer_(computer_programming)) to it, and one of those pointers is dereferenced. By then the memory may have been reassigned to another use, with unpredictable results.
* *Double free bugs*, which occur when the program tries to free a region of memory that has already been freed, and perhaps already been allocated again.
* Certain kinds of [*memory leaks*](https://en.wikipedia.org/wiki/Memory_leak), in which a program fails to free memory occupied by objects that have become [unreachable](https://en.wikipedia.org/wiki/Unreachable), which can lead to memory exhaustion. (Garbage collection typically does not deal with the unbounded accumulation of data that is reachable, but that will actually not be used by the program.)
* Efficient implementations of [persistent data structures](https://en.wikipedia.org/wiki/Persistent_data_structure)

Some of the bugs addressed by garbage collection have security implications.

### == Disadvantages Typically, garbage collection has certain disadvantages, including consuming additional resources, performance impacts, possible stalls in program execution, and incompatibility with manual resource management. Garbage collection consumes computing resources in deciding which memory to free, even though the programmer may have already known this information. The penalty for the convenience of not annotating object lifetime manually in the source code is [overhead](https://en.wikipedia.org/wiki/Overhead_(computing)), which can lead to decreased or uneven performance.[[4]](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)#cite_note-zorn1993-4) A peer-reviewed paper came to the conclusion that GC needs five times the memory to compensate for this overhead and to perform as fast as explicit memory management.[[5]](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)#cite_note-5) Interaction with memory hierarchy effects can make this overhead intolerable in circumstances that are hard to predict or to detect in routine testing. The impact on performance was also given by Apple as a reason for not adopting garbage collection in [iOS](https://en.wikipedia.org/wiki/IOS) despite being the most desired feature.[[6]](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)#cite_note-wwdc2011-6)

**🡺**[**Why Java and Python garbage collection methods are different?**](https://stackoverflow.com/questions/21934/why-java-and-python-garbage-collection-methods-are-different)

**PYTHON**

There are drawbacks of using reference counting. One of the most mentioned is circular references: Suppose A references B, B references C and C references B. If A were to drop its reference to B, both B and C will still have a reference count of 1 and won't be deleted with traditional reference counting. CPython (reference counting is not part of python itself, but part of the C implementation thereof) catches circular references with a separate garbage collection routine that it runs periodically...

Another drawback: Reference counting can make execution slower. Each time an object is referenced and dereferenced, the interpreter/VM must check to see if the count has gone down to 0 (and then deallocate if it did). Garbage Collection does not need to do this.

Also, Garbage Collection can be done in a separate thread (though it can be a bit tricky). On machines with lots of RAM and for processes that use memory only slowly, you might not want to be doing GC at all! Reference counting would be a bit of a drawback there in terms of performance...

There are two broad approaches for tracking down dead objects: tracing and reference counting. In tracing the GC starts from the "roots" - things like stack references, and traces all reachable (live) objects. Anything that can't be reached is considered dead. In reference counting each time a reference is modified the object's involved have their count updated. Any object whose reference count gets set to zero is considered dead.

With basically all GC implementations there are trade offs but tracing is usually good for high through put (i.e. fast) operation but has longer pause times (larger gaps where the UI or program may freeze up). Reference counting can operate in smaller chunks but will be slower overall. It may mean less freezes but poorer performance overall.

Additionally, a reference counting GC requires a cycle detector to clean up any objects in a cycle that won't be caught by their reference count alone. Perl 5 didn't have a cycle detector in its GC implementation and could leak memory that was cyclic.

best garbage collector is the one with the best performance, which currently seems to be the Java-style generational garbage collectors that can run in a separate thread and has all these crazy optimizations, etc.

**🡺 IN JAVA**

The [garbage collector](http://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29) is a program which runs on the [Java Virtual Machine](http://en.wikipedia.org/wiki/Java_Virtual_Machine) which gets rid of objects which are not being used by a Java application anymore. It is a form of *automatic memory management*.

When a typical Java application is running, it is creating new objects, such as Strings and Files, but after a certain time, those objects are not used anymore. For example, take a look at the following code:

for (File f : files) {

String s = f.getName();

}

In the above code, the String s is being created on each iteration of the for loop. This means that in every iteration, a little bit of memory is being allocated to make a String object.

Going back to the code, we can see that once a single iteration is executed, in the next iteration, the String object that was created in the previous iteration is not being used anymore -- that object is now considered "garbage".

Eventually, we'll start getting a lot of garbage, and memory will be used for objects which aren't being used anymore. If this keeps going on, eventually the Java Virtual Machine will run out of space to make new objects.

That's where the garbage collector steps in.

The garbage collector will look for objects which aren't being used anymore, and gets rid of them, freeing up the memory so other new objects can use that piece of memory.

In Java, memory management is taken care of by the garbage collector, but in other languages such as C, one needs to perform memory management on their own using functions such as [mallocand free](http://en.wikipedia.org/wiki/Malloc). [Memory management](http://en.wikipedia.org/wiki/Memory_management) is one of those things which are easy to make mistakes, which can lead to what are called [memory leaks](http://en.wikipedia.org/wiki/Memory_leak) -- places where memory is not reclaimed when they are not in use anymore.

Automatic memory management schemes like garbage collection makes it so the programmer does not have to worry so much about memory management issues, so he or she can focus more on developing the applications they need to develop.

**🡺Conservative Type GC**

A garbage collector must scan all objects and invocations (execution stack) to identify all of the "live" addresses in the executing program and then "collect" objects that do not have "live" addresses. In some environments it's possible for the GC algorithm to be PRECISE and know exactly what is an object address and what is not. In other environments it must scan parts of storage (most notably the execution stack) where there are words of storage that MIGHT be an object address and make the CONSERVATIVE assumption that if it looks like a valid address, and there is an object that has that address, then the object should not be collected.  
There are advantages to conservative collection, most notably that the code generator (if not interpreted) is freer to allocate variables where and when it needs them and it need not keep rigorous track of which are object pointers. (The need to keep track of object pointer locations can lead to less well optimized code, in addition to making the code generator considerably more complex. Also, a conservative collector stands some reasonable chance of being used with a compiler which was never intended to support garbage collection, while a precise collector would require that the compiler be radically altered.)  
The major disadvantage of the conservative approach is that a full "copying" collector cannot be implemented. When copying is done the pointers to the copied objects must be updated, and if it's not clear whether a given bit value is an object pointer or just a numeric value, it cannot be safely determined whether or not it should be modified when the object is copied. There's also the disadvantage that some "dead" objects may end up not getting collected, due to random bit patterns that look like their addresses, though in practice this is not a serious concern.

**🡺 What are the differences between Python and Java memory management?**

**PYTHON**

1) Python memory management uses a heap .  
2) All the objects declared and the data structures involved are contained in the private heap. The heap management is done via the use of python memory manager and is performed by the interpreter itself and that the user has no control over it.  
3)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.  
 **🡺 JAVA MEMORY MANAGEMENT :-**1) In Java, memory is allocated only to objects. There is no explicit allocation of memory, there is only the creation of new objects.  
2) JRT (java run time) provides us with a garbage collector which is responsible for freeing the space which no longer is having any reference to an object.

In C/Java language, **int a=12**means   
Step 1: 2 bytes of memory is reserved, as it is ***int***type Step 2: variable ***a***is assigned to that memory location.Step 3: Value **12**is placed in that location.In Python, **a=12**means  
Step 1: object ***12***is created.  
Step 2: variable ***a***is **labelled** to that object.

In python, the objects are labelled and referenced by the variables, and once the reference count becomes zero, that object is garbage collected.  
ex:a=12 b=a c=b Now, if you check the values,   
>>> a=12 >>> b=a >>> c=b  >>> a,b,c (12, 12, 12)  It means that object **12**has THREE variables referenced. Let us overwrite c=34, a=23 Then, reference count will be ONE. Now, let us delete **a** del aNow, there is no reference to 12.   
So, in the next clock cycle, the garbage collector removes this object from the memory. In this way, python optimizes the memory usage

**🡺 What are the similarities between Prolog and Java?**

Prolog is a logic language where you attempt to prove facts over universally qualified variables; Java is an imperative language where you instruct the system to perform operations directly.Prolog supports two categories of values: *atoms* and *functors*. Certain atoms may be interpreted by clauses as having additional meaning (e.g., integers) but for the most part that is all you have. Java has a richer set of primitives and its primary compound is the *class*.Prolog variables are untyped and are immutable once bound. Java variables (unless marked final) are mutable.

Both allow recursion, although in Java loop constructs are preferable for performance and memory issues. Both have lexical scoping. Java allows variables with global scope; Prolog has a global database where clauses can be added and retracted during execution. Global data is considered bad style in both languages.The unit of computation in Java is the *basic block*: the IP starts at the top of the block, works its way through it, and exits at the bottom. A Java *method* consists of basic blocks linked together with various conditional flow controls. Prolog has *boxes*: depending on the system, the box may have four to nine *ports*. On a basic level, a Prolog box has two incoming ports (“call” and “redo”) and two outgoing ports (“exit” and “fail”). A Prolog *clause* consists of boxes with their ports connected in various configurations.

Words: unquoted words in Java must be valid keywords or defined variables; in Prolog, any word that begins with a uppercase letter or underscore is automatically a variable, any word that begins with a number or lowercase letter is an atom. Atoms may also be arbitrary strings in single quotes.Prolog allows you to create new operators and define their precedence. You can be evil and change the precedence of existing operators too. These are all sugaring rules: internally, Prolog treats expressions as functors.Prolog usually is used with its REPL. Java is compiled.

**🡺 Parallelism is Java**

Parallel computing involves dividing a problem into subproblems, solving those problems simultaneously (in parallel, with each subproblem running in a separate thread), and then combining the results of the solutions to the subproblems. Java SE provides the [fork/join framework](https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html), which enables you to more easily implement parallel computing in your applications. However, with this framework, you must specify how the problems are subdivided (partitioned). With aggregate operations, the Java runtime performs this partitioning and combining of solutions for you.  
One difficulty in implementing parallelism in applications that use collections is that collections are not thread-safe, which means that multiple threads cannot manipulate a collection without introducing [thread interference](https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html) or [memory consistency errors](https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html). The Collections Framework provides [synchronization wrappers](https://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html), which add automatic synchronization to an arbitrary collection, making it thread-safe. However, synchronization introduces [thread contention](https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html#thread_contention). You want to avoid thread contention because it prevents threads from running in parallel. Aggregate operations and parallel streams enable you to implement parallelism with non-thread-safe collections provided that you do not modify the collection while you are operating on it.  
Note that parallelism is not automatically faster than performing operations serially, although it can be if you have enough data and processor cores. While aggregate operations enable you to more easily implement parallelism, it is still your responsibility to determine if your application is suitable for parallelism.

**🡺 Type checking in JAVA**Java is a statically typed language, so the compiler does most of this checking for you. Once you declare a variable to be a certain type, the compiler will ensure that it is only ever assigned values of that type (or values that are sub-types of that type).

The examples you gave (int, array, double) these are all primitives, and there are no sub-types of them. Thus, if you declare a variable to be an int:

int x;

You can be sure it will only ever hold int values.

If you declared a variable to be a List, however, it is possible that the variable will hold sub-types of List. Examples of these include ArrayList, LinkedList, etc.

If you did have a List variable, and you needed to know if it was an ArrayList, you could do the following:

List y;

...

if (y instanceof ArrayList) {

...its and ArrayList...

}

However, if you find yourself thinking you need to do that, you may want to re-think your approach. In most cases, if you follow object-oriented principals, you will not need to do this. There are, of course, exceptions to every rule, though.

# 🡺[Difference between volatile and synchronized in Java](https://stackoverflow.com/questions/3519664/difference-between-volatile-and-synchronized-in-java)

It's important to understand that there are two aspects to thread safety: (1) execution control, and (2) memory visibility. The first has to do with controlling when code executes (including the order in which instructions are executed) and whether it can execute concurrently, and the second to do with when the effects in memory of what has been done are visible to other threads. Because each CPU has several levels of cache between it and main memory, threads running on different CPUs or cores can see "memory" differently at any given moment in time because threads are permitted to obtain and work on private copies of main memory.

Using synchronized prevents any other thread from obtaining the monitor (or lock) **for the same object**, thereby preventing all code blocks protected by synchronization **on the same object** from executing concurrently. Synchronization also creates a "happens-before" memory barrier, causing a memory visibility constraint such that anything done up to the point some thread releases a lock appears to another thread subsequently acquiring **the same lock** to have happened before it acquired the lock. In practical terms, on current hardware, this typically causes flushing of the CPU caches when a monitor is acquired and writes to main memory when it is released, both of which are (relatively) expensive.

Using volatile, on the other hand, forces all accesses (read or write) to the volatile variable to occur to main memory, effectively keeping the volatile variable out of CPU caches. This can be useful for some actions where it is simply required that visibility of the variable be correct and order of accesses is not important. Using volatile also changes treatment of long and double to require accesses to them to be atomic; on some (older) hardware this might require locks, though not on modern 64 bit hardware. Under the new (JSR-133) memory model for Java 5+, the semantics of volatile have been strengthened to be almost as strong as synchronized with respect to memory visibility and instruction ordering (see <http://www.cs.umd.edu/users/pugh/java/memoryModel/jsr-133-faq.html#volatile>). For the purposes of visibility, each access to a volatile field acts like half a synchronization.

So, now both forms of memory barrier (under the current JMM) cause an instruction re-ordering barrier which prevents the compiler or run-time from re-ordering instructions across the barrier. In the old JMM, volatile did not prevent re-ordering. This can be important, because apart from memory barriers the only limitation imposed is that,  *for any particular thread*, the net effect of the code is the same as it would be if the instructions were executed in precisely the order in which they appear in the source.

Volatile variables are useful only when ***all*** operations performed on them are "atomic", such as my example where a reference to a fully formed object is only read or written (and, indeed, typically it's only written from a single point). Another example would be a volatile array reference backing a copy-on-write list, provided the array was only read by first taking a local copy of the reference to it.

**PYTHON**

🡺 **PYTHON 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.

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. 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.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if she regularly manipulates 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.

So if you want to hold on to an object, just hold a reference to it. If you want the object to be freed (eventually) remove any references to it.

def foo(names):

for name in names:

print name

foo(["Eric", "Ernie", "Bert"])

foo(["Guthtrie", "Eddie", "Al"])

Each of these calls to foo creates a Python list object initialized with three values. For the duration of the foo call they are referenced by the variable names, but as soon as that function exits no variable is holding a reference to them and they are fair game for the garbage collector to delete.

**🡺 What is Namespace in Python**Namespace is a way to implement scope.  
In Java (or C) the compiler determines where a variable is visible through static scope analysis.

* In C, scope is either the body of a function or it's global or it's external. The compiler reasons this out for you and resolves each variable name based on scope rules. External names are resolved by the linker after all the modules are compiled.
* In Java, scope is the body of a method function, or all the methods of a class. Some class names have a module-level scope, also. Again, the compiler figures this out at compile time and resolves each name based on the scope rules.

In Python, each package, module, class, function and method function owns a "namespace" in which variable names are resolved. Plus there's a global namespace that's used if the name isn't in the local namespace.  
Each variable name is checked in the local namespace (the body of the function, the module, etc.), and then checked in the global namespace.  
Variables are generally created only in a local namespace. The global and nonlocal statements can create variables in other than the local namespace.  
When a function, method function, module or package is evaluated (that is, starts execution) a namespace is created. Think of it as an "evaluation context". When a function or method function, etc., finishes execution, the namespace is dropped. The variables are dropped. The objects may be dropped, also.

**PROLOG**

[Prolog](http://en.wikipedia.org/wiki/Prolog) is a declarative or logic language created in 1972.  
It works a little like SQL: You give it some facts and ask a question, and, without specifying how, prolog will find the results for you. It can express a lot of things that you cannot express in SQL.  
A relational database that can run SQL is a complicated program, but Prolog is very simple and works using 2 simple principles:  
 Unification

Backtracking

The Japanese [Fifth Generation Program](http://en.wikipedia.org/wiki/Fifth_Generation_Computer_Systems_project) was built in Prolog. That was a big deal and scared many people in the West in the 1980s

**🡺 Prolog uses Dynamic Scoping**

First things first: does prolog uses dynamic scoping? No, it does not, and the reason why it doesn't is because prolog needs to keep track of the variables defined, so that any new assignment would refer to the initial variable -- and not to last value assigned to that variable name.

Now, as you may be aware of, you're not able to this in prolog:

#!/usr/bin/swipl

my\_first\_I(I) :- I is 10.

my\_second\_I(I) :- I is 100.

test(I) :-

I is 1,

write(I), nl,

my\_first\_I(\_),

write(I), nl,

my\_second\_I(\_),

write(I), nl.

Output:

?- test(I).

1

1

1

I = 1.

This example does simply point out that, in prolog, the variable is defined in its own local scope, with no leakage into another function's definition.

Now, the second part: how does the ! work? The logic programming solutions generated by Prolog are derived after the application of an [unification algorithm](http://en.wikipedia.org/wiki/Unification_%28computer_science%29). During the process, a tree-like structure is built in a DFS fashion, performing evaluations on the statements according to the values variables may assume. After each leaf-level expansion, Prolog backtracks up to the next available expansion.

If an ! is found, the breadth expansion is stopped, cutting off other possible values for the local solution, and only perfoming the deeper expansions in the tree. [Here](http://en.wikibooks.org/wiki/Prolog/Cuts_and_Negation) is a quite simple usage of the ! operator.

**🡺 Prolog Memory Management**Actual performance of Prolog based applications largely depends on how the underlying system implements memory management. Most Prolog systems are based on the WAM, which is built around a set of stacks. The WAM is highly optimized to recover memory on backtracking and on tail-recursive predicates. Still, deterministic computations can create intermediate structures that can only be freed through garbage collection.

 The traditional Marseille Prolog used a single stack for the management of Prologdata areas. More efficient systems allow recovering memory during forward execution by dividing this stack into three or four stacks. In the WAM these are the Heap, the Local (or environment plus choice-point) stack, and the Trail. The Local Stack.  The Local stack includes environments and choice-points. Environments correspond to live clauses, and are the equivalent of activation records on imperative languages. They have two control fields in the WAM, plus a clause dependent number of slots for permanent variables, that is, for variables that were created in the body of the clause and that span several sub-goals. The WAM provides several optimizations for deterministic computations such as last call optimization and environment trimming. Choice-points correspond to unexplored clauses in the search tree. They are created for non-determinate goals, that is, for goals where we cannot prove at entry that at most one clause matches. In database terminology they are a snapshot: we are torecover the state of a computation at the moment of goal entry from the data in the choice-point. Choice-points include the current stack tops, program pointers, and the arguments for the current goal. Choice-points are allocated at clause entry and are recovered at backtracking or after execution of a cut. Cut is quite important, as support for last call optimization also guarantees that all environments created

**Is Prolog Static/Dynamic or Untyped?**

Prolog is mostly untyped in the sense that you can pass any kind of term to any predicate and, usually, the worst case is that the predicate will not succeed. However, arithmetic predicates, such as is and =:= expect numeric arguments and may blow up - so there is a notion of types there.

Non-pure predicates might also expect objects of type "file handle" and blow up otherwise.  
So, calling Prolog "untyped" is not strictly true.

**Why Gprolog use more RAM?**In your ps/top-listing there are two numbers: The virtual memory allocated (size) and the physical memory that is actually in use (res). From this it seems that GNU Prolog initially uses less memory than SWI. That is: 2196K for GNU versus 3728K for SWI.But you cannot conclude anything of relevance from these numbers alone. The only thing you can say is that the default-environment with the toplevel needs so much memory to start up — provided you have not "paged out" the processes with another program...Both systems try to keep memory consumption low, but on different levels:

[GNU Prolog](http://www.gprolog.org/) offers compilation to stand-alone executables that skip unused built-in predicates. The executable code is treated similarly to C: It is thus read-only mapped into physical memory. If you run several instances of such an executable, they all will share the same physical memory for the executables.

On the downside, GNU Prolog lacks garbage collection. Both for the heap (copystack) and for atoms (symboltable). To avoid overflow handling, memory areas are allocated generously. But this is only a reservation for virtual memory. As far as I know all current Unix variations over-commit virtual memory so this does not take away the corresponding swap-space.

[SWI-Prolog](http://www.swi-prolog.org/) on the other hand allocates its Prolog code in writeable memory. Further that memory is "touched" (marked/unmarked) during GC. As a consequence Prolog programs cannot be shared between different SWI instances, not even with [dynamic re-sharing like mergemem](http://www.complang.tuwien.ac.at/ulrich/mergemem/). That is, mergemem (or similar) can page-share it, but upon the next db-GC it is copy-on-write unshared. See the link how sharing can reduce memory consumption for SICStus. But then SWI has multithread-support which somewhat induces sharing.

SWI boasts one of the very best and complete garbage collectors for the heap and atoms.

🡺**Unification in Prolog**Yes, it does unify, and it does so because g(Y) is a term to be evaluated, as well as a -- in the first example you pointed.

You can check the evaluation in a prolog interpreter:

?- f(g(Y),h(c,d)) = f(X,h(W,d)).

X = g(Y),

W = c.

The unification process works in a depth-first fashion, unifying members and returning each of the available answer, until no further combination is possible.

This means the **unification method** is called for f(g(Y),h(c,d)) = f(X,h(W,d)), that finds out the available matchings: g(Y) = X, h(c, d) = h(W, d).

Then, the unification is performed upon g(Y) = X, that, since there's no further possible reduction, returns X = g(Y).

Then, the same method is called upon the matching h(c, d) = h(W, d), which gives you c = W, and no other matching, resulting, thus, in W = c.

The answers, after unification, are returned, and it's usually returned false to point when no matching/further matching is possible.

As pointed by CapelliC, the variable Y, after the unification process, is still unbound. The unification is performed upon unbound variables, which means:

* the unification of h(c, d) = h(W, d) returns h(\_) = h(\_), and this allows the unification to continue, since h is a term, and not an unbound var;
* the unification of d = d is a matching of terms, and does not form an attribution -- or binding;
* the unification of c = W forms an attribution, and the variable W is bound to the term c, since it was not bound before -- a comparison would be performed otherwise;
* the unification of X = g(Y) simply binds the unbound variable X to the term g(Y), and g(Y) is a term with an unbound variable, since there's no available unification to g(Y).

**SCHEME and LISP**

🡺 **disadvantages of reference counting schemes**:  
- Each object needs to store a reference count. In other words there's a word overhead for every object. Programs use more memory, and are consequently slower because they are more likely to fill up the cache or spill into swap.  
- Reference counting is expensive - every time you manipulate pointers to an object you need to update and check the reference count. Pointer manipulation is frequent, so this slows your program and bloats the code size of compiled code.  
- They cannot collect so-called circular, or self-referential structures. I've programmed in many languages in many years and can't recall ever having created one of these.  
- Graph algorithms, of course, violate the previous assumption. Don't try to implement the TSP in Perl!

**🡺 GC in Scheme**In languages like C or Pascal, data objects may be allocated in several ways. (Recall that by "objects" I just mean data objects like records.) They may be allocated statically (as in the case of global variables), or on an activation stack as part of a procedure activation record (as in the case of local variables), or dynamically allocated on the heap at run time using an alloction routine like malloc or new.  
Scheme is simpler--all objects are allocated on the heap, and referred to via pointers. The Scheme heap is garbage collected, meaning that the Scheme system automatically cleans up after you. Every now and then, the system figures out which objects aren't in use anymore, and reclaims their storage. (This determination is very conservative and safe--the collector will never take back any object that your program holds a pointer to, or might reach via any path of pointer traversals. Don't be afraid that the collector will eat objects you still care about while you're not looking!)  
The use of garbage collection supports the abstraction of indefinite extent. That means that all objects conceptually live forever, or at least as long as they might matter to the program--there's no concept (at the language level) of reusing memory. From the point of view of a running program, memory is infinite--it can keep allocating objects indefinitely, without ever reusing their space.  
Of course, this abstraction breaks down if there really isn't enough memory for what you're trying to do. If you really try to create data structures that are bigger than the available memory, you'll run out. Garbage collection can't give you memory you don't have.

**🡺 Continuation in Scheme**

(call/cc

(lambda (k)

(\* 5 (k 4)))) => 4

Forget about call/cc for a moment. Every expression/statement, in any programming language, has a continuation - which is, what you do with the result. In C, for example,

x = (1 + (2 \* 3));

printf ("Done");

has the continuation of the math assignment being printf(...); the continuation of (2 \* 3) is 'add 1; assign to x; printf(...)'. Conceptually the continuation is there whether or not you have access to it. Think for a moment what information you need for the continuation - the information is 1) the heap memory state (in general), 2) the stack, 3) any registers and 4) the program counter.

So continuations exist but usually they are only implicit and can't be accessed.  
In Scheme, and a few other languages, you have access to the continuation. Essentially, behind your back, the compiler+runtime bundles up all the information needed for a continuation, stores it (generally in the heap) and gives you a handle to it. The handle you get is the function 'k' - if you call that function you will continue exactly after the call/cc point. Importantly, you can call that function multiple times and you will always continue after the call/cc point.  
Let's look at some examples:  
> (+ 2 (call/cc (lambda (cont) 3)))  
5  
In the above, the result of call/cc is the result of the lambda which is 3. The continuation wasn't invoked.  
Now let's invoke the continuation:

> (+ 2 (call/cc (lambda (cont) (cont 10) 3)))

12

By invoking the continuation we skip anything after the invocation and continue right at the call/ccpoint. With (cont 10) the continuation returns 10 which is added to 2 for 12.

Now let's save the continuation.

> (define add-2 #f)

> (+ 2 (call/cc (lambda (cont) (set! add-2 cont) 3)))

5

> (add-2 10)

12

> (add-2 100)

102

By saving the continuation we can use it as we please to 'jump back to' whatever computation followed the call/cc point.

Often continuations are used for a non-local exit. Think of a function that is going to return a list unless there is some problem at which point '() will be returned.

(define (hairy-list-function list)

(call/cc

(lambda (cont)

;; process the list ...

(when (a-problem-arises? ...)

(cont '()))

;; continue processing the list ...

value-to-return)))

# 🡺 [What's the difference between lists constructed by quote and those constructed by cons in Scheme?](https://softwareengineering.stackexchange.com/questions/231604/whats-the-difference-between-lists-constructed-by-quote-and-those-constructed-b)

E.g. if you have a file which contains  
(define l1 '(1 2 3))  
(define l2 '(4 2 3))  
then the compiler is permitted to allocate l1 and l2 in a way that they share their common tail (cdr l1) and (cdr l2) and/or in the read-only memory.  
Modification of such lists is undefined behavior. Do not do it.  
list

list and cons create fresh objects (different from everything which already exist), they allocate and populate memory. You own them - you can modify them as much as you want.

🡺 **Different between SCHEME and LISP  
 Scheme**

is a language built on the principle of providing an elegant, consistent, well thought-through base language substrate which both practical and academic application languages can be built upon.

Rarely will you find someone writing an application in pure R5RS (or R6RS) Scheme, and because of the minimalistic standard, most code is not portable across Scheme implementations. Most Scheme dialects and libraries focus on functional programming idioms like recursion instead of iteration. There are various object systems you can load as libraries when you want to do OOP, but integration with existing code heavily depends on the Scheme dialect and its surrounding culture (Chicken Scheme seems to be more object-oriented than Racket, for instance).  
Interactive programming is another point that Scheme subcommunities differ in. MIT Scheme is known for strong interactivitiy support, while PLT Racket feels much more static. In any case, interactive programming does not seem to be a central concern to most Scheme subcommunities, and I have yet to see a programming environment similarly interactive as most Common Lisps'.

**Common Lisp**

 is a battle-worn language designed for practical programming. It is full of ugly warts and compatibility hacks -- quite the opposite of Scheme's elegant minimalism. But it is also much more featureful when taken for itself.

Common Lisp has bred a relatively large ecosystem of portable libraries. You can usually switch implementations at any time, even after application deployment, without too much trouble. Overall, Common Lisp is much more uniform than Scheme, and more radical language experiments, if done at all, are usually embedded as a portable library rather than defining a whole new language dialect. Because of this, language extensions tend to be more conservative, but also more combinable (and often optional).  
Both because of the image-based, interactive development, and because of the larger language, Lisp implementations are usually less portable across operating systems than Scheme implementations are. Getting a Common Lisp to run on an embedded device is not for the faint of heart, for example. Similarly to the Java Virtual Machine, you also tend to encounter problems on machines where virtual memory is restricted (like OpenVZ-based virtual servers). Scheme implementations, on the other hand, tend to be more compact and portable. The increasing quality of the ECL implementation has mitigated this point somewhat, though its essence is still true.  
If you care for commercial support, there are a couple of companies that provide their own Common Lisp implementations including graphical GUI builders, specialized database systems, et cetera.

**Summing up**,   
Scheme is a more elegantly designed language. It is primarily a functional language with some dynamic features. Its implementations represent various incompatible dialects with distinctive features. Common Lisp is a fully-fledged, highly dynamic, multi-paradigm language with various ugly but pragmatic features, whose implementations are largely compatible with one another. Scheme dialects tend to be more static and less interactive than Common Lisp; Common Lisp implementations tend to be heavier and trickier to install.

🡺**Different between let\* and let in scheme**

Use let\* instead of let.

The difference between let and let\* is the following:

let\* binds variables from left to right. Earlier bindings can be used in new binding further to the right (or down).

let on the other hand can be thought of as syntactic sugar (or macro) for simple lambda abstraction:

(let ((a exp1)

(b exp2))

exp)

is equivalent to   
((lambda (a b)  
 exp)  
exp1 exp2

## **🡺 Equality and Identity: equal?, eqv?, eq? in Scheme --** eq? is pointer comparison. It returns #t iff its arguments literally refer to the same objects in memory. Symbols are unique ('fred always evaluates to the same object). Two symbols that look the same are eq. Two variables that refer to the same object are eq.  -- eqv? is like eq? but does the right thing when comparing numbers. eqv? returns #t iff its arguments are eq *or* if its arguments are numbers that have the same value.  eqv? does not convert integers to floats when comparing integers and floats though. -- equal? returns true if its arguments have the same structure. Formally, we can define equal? recursively.  equal? returns #t iff its arguments are eqv, or if its arguments are lists whose corresponding elements are equal (note the recursion). Two objects that are eq are both eqv and equal. Two objects that are eqv are equal, but not necessarily eq. Two objects that are equal are not necessarily eqv or eq.  eq is sometimes called an identity comparison and equal is called an equality comparison.

**LOGICAL**

**🡺 Compiled language Vs Interpreted Language**

A compiled language is one where the program, once compiled, is expressed in the instructions of the target machine. For example, an addition "+" operation in your source code could be translated directly to the "ADD" instruction in machine code.

An interpreted language is one where the instructions are not directly executed by the target machine, but instead read and executed by some other program (which normally *is* written in the language of the native machine). For example, the same "+" operation would be recognised by the interpreter at run time, which would then call its own "add(a,b)" function with the appropriate arguments, which would then execute the machine code "ADD" instruction.

You can do anything that you can do in an interpreted language in a compiled language and vice-versa - they are both Turing complete. Both however have advantages and disadvantages for implementation and use.

advantages of compiled languages:  
[1] Faster performance by directly using the native code of the target machine   
[2] Opportunity to apply quite powerful optimizations during the compile stage  
 advantages of interpreted languages:  
-[1] Easier to implement (writing good compilers is very hard!!)  
- [2] No need to run a compilation stage: can execute code directly "on the fly"   
-[3] Can be more convenient for dynamic languages  
Note that modern techniques such as bytecode compilation add some extra complexity - what happens here is that the compiler targets a "virtual machine" which is not the same as the underlying hardware. These virtual machine instructions can then be compiled again at a later stage to get native code (e.g. as done by the Java JVM JIT compiler)

**🡺 Why first OOP programming (Simula67) is not famous**

Some might say the OOP crowd is more trendy. A better reason is that OOP languages like C++ and Java have much more dependence on external libraries, due to the fact that OOP concepts like classes, subclassing, etc, make it very easy to create and maintain external modules. But as time passes, people will begin to write their modules in more modern OOP languages, which causes earlier OOP languages to be unusable due to lack of library support.

## 🡺 **(Static) Scope and (Dynamic) Extent**

Scope is a property of the static world (i.e. of the written program, so that it all gets sorted out at compile time).   
An identifier which is declared at some point in a program (the **defining occurrence**) may be used, with its declared meaning, at other points in the program (**applied occurrences**). The scope of the identifier is the region of the text of that program in which those applied occurrences may occur.  
The static environment must change at the start of the scope of an identifier and again at the end of that scope. In some languages many identifiers start their scope at exactly the same place (e.g. the start of a function), in others each declaration starts a fresh scope. A scope may continue until the end of the subprogram or block containing the declaration, or whatever. In most languages the same identifier can be used in several declarations, each with its own scope.  
Extent is the corresponding property of the dynamic world (i.e. it gets sorted out at run time).   
If a location(s) for a variable is provided at some time, and if later on that location(s) is taken away again (perhaps for use by a different variable), then the interval of time over which the location(s) for that variable existed is the extent of that variable.   
The dynamic environment must change when that variable appears and again when it vanishes. It may be that a variable, once allocated, remains around until the end of the program (its extent is the lifetime of the program), or it may be that the variable is only around during the running of some procedure (its extent is the lifetime of that procedure). If such a procedure manages to call itself recursively, then there may be several such variables created, each with its own extent (and these extents are nested inside each other).  
In simpler languages (such as PASCAL), the extent of a variable corresponds so closely to the scope of the identifier which accesses it that language manuals sometimes do not distinguish between the two concepts.

🡺 **GC vs. reference counting**

Perl has a form of garbage collection, but it uses a simple scheme called **reference counting**. Simply put, each Perl object keeps a count of the number of other objects pointing (referencing) itself. When the count falls to zero, nothing is pointing at this object, and so the object can be freed.  
Reference counting is not considered as serious garbage collection by computer scientists,   
yet it has one big **practical advantage over full garbage collectors.**   
With reference counting, you can avoid many explicit calls to close/closedir in code. For example:  
**let** read\_file filename =  
**let** chan = open\_in filename **in**  
 *(\* read from chan \*)* **in**  
List.iter read\_file files  
Calls to read\_file open the file but don't close it. Because OCaml uses a full garbage collector chan isn't collected until some time later when the minor heap becomes full. In addition, **OCaml will not close the channel when it collects the handle's memory**. So this program would eventually run out of file descriptors.  
You need to be aware of this when writing OCaml code which uses files or directories or any other heavyweight object with complex finalisation.  
  
**🡺 The main disadvantages to using a garbage collector, in my opinion, are:  
[1]** Non-deterministic cleanup of resources. Sometimes, it is handy to say "I'm done with this, and I want it cleaned NOW". With a GC, this typically means forcing the GC to cleanup everything, or just wait until it's ready - both of which take away some control from you as a developer.  
[2] Potential performance issues which arise from non-deterministic operation of the GC. When the GC collects, it's common to see (small) hangs, etc. This can be particularly problematic for things such as real-time simulations or games.

**🡺 Difference between Functional and Imperative, Declarative Programming language**

**Definition:** An imperative language uses a sequence of statements to determine how to reach a certain goal. These statements are said to change the state of the program as each one is executed in turn.

**Examples:** Java is an imperative language. For example, a program can be created to add a series of numbers:

int total = 0;

int number1 = 5;

int number2 = 10;

int number3 = 15;

total = number1 + number2 + number3;

Each statement changes the state of the program, from assigning values to each variable to the final addition of those values. Using a sequence of five statements the program is explicitly told how to add the numbers 5, 10 and 15 together.

**Functional languages:** The functional programming paradigm was explicitly created to support a pure functional approach to problem solving. Functional programming is a form of declarative programming.

**Advantages of Pure Functions:** The primary reason to implement functional transformations as pure functions is that pure functions are composable: that is, self-contained and stateless. These characteristics bring a number of benefits, including the following: Increased readability and maintainability. This is because each function is designed to accomplish a specific task given its arguments. The function does not rely on any external state.

Easier reiterative development. Because the code is easier to refactor, changes to design are often easier to implement. For example, suppose you write a complicated transformation, and then realize that some code is repeated several times in the transformation. If you refactor through a pure method, you can call your pure method at will without worrying about side effects.

Easier testing and debugging. Because pure functions can more easily be tested in isolation, you can write test code that calls the pure function with typical values, valid edge cases, and invalid edge cases.

**For OOP People or Imperative languages:**

Object-oriented languages are good when you have a fixed set of operations on things and as your code evolves, you primarily add new things. This can be accomplished by adding new classes which implement existing methods and the existing classes are left alone.

Functional languages are good when you have a fixed set of things and as your code evolves, you primarily add new operations on existing things. This can be accomplished by adding new functions which compute with existing data types and the existing functions are left alone.

Cons:

It depends on the user requirements to choose the way of programming, so there is harm only when users don’t choose the proper way.

When evolution goes the wrong way, you have problems:

1. Adding a new operation to an object-oriented program may require editing many class definitions to add a new method
2. Adding a new kind of thing to a functional program may require editing many function definitions to add a new case

**Declarative Programming Language**

In computer science, declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow. It attempts to minimize or eliminate side effects by describing what the program should accomplish, rather than describing how to go about accomplishing it. This is in contrast from imperative programming, which requires a detailed description of the algorithm to be run.

Declarative programming considers programs as theories of a formal logic, and computations as deductions in that logic space. Declarative programming has become of particular interest recently, as it may greatly simplify writing parallel programs.

🡺 **Static storage / Stack-Based Storage / Heap-Based Storage allocation**

**STATIC STORAGE**

Static storage allocation is appropriate when the storage requirements are known at compile time. For a compiled, linked language, the compiler can include the specific memory address for the variable or constant in the code it generates. (This may be adjusted by an offset at link time.) Examples:

* code in languages without dynamic compilation
* all variables in FORTRAN IV
* global variables in C, Ada, Algol
* constants in C, Ada, Algol

**STACK-BASED Storage Allocation**

Stack-based storage allocation is appropriate when the storage requirements are not known at compile time, but the requests obey a last-in, first-out discipline. Examples:

* local variables in a procedure in C/C++, Ada, Algol, or Pascal
* procedure call information (return address etc).

Stack-based allocation is normally used in C/C++, Ada, Algol, and Pascal for local variables in a procedure and for procedure call information. It allows for recursive procedures, and also allocates data only when the procedure or function has been called -- but is reasonably efficient at the same time.

Java uses a stack for allocating frames for each method invocation. Parameters and local variables are stored in the stack frame. Primitive types are contained within the stack frame itself. Object references are stored in the stack frame, and point to the actual object storage in the heap. Example:

import java.awt.Point;

class Squid {

public static void main(String[] args) {

int n = 1;

Point p = new Point(10,20);

Point q;

q = test(n,p);

}

public Point test(int i, Point r) {

Point s;

s = new Point(r.x+i, r.y+i);

return s;

}

}

Here, n and i will be stored in the stack frames. The points will be stored in the heap.

Typically, in a compiled language like C++ or Algol, a pointer to the base of the current stack frame is held in a register, say R0. A reference to a local scalar variable can be compiled as a load of the contents of R0 plus a fixed offset. Note that this relies on the data having known size. To compile a reference to a dynamically sized object, e.g. an array, use indirection. The stack contains an array descriptor, of fixed size, at a known offset from the base of the current stack frame. The descriptor then contains the actual address of the array, in addition to bounds information.There are two important limitations to pure stack-based storage allocation. First, the only way to return data from a procedure or function is to copy it -- if you try to simply return a reference to it, the storage for the date will have vanished after the procedure or function returns. This isn't an issue for scalar data (integers, floats, booleans), but is an issue for large objects such as arrays. For this reason you can't, for example, return a locally declared array from a function in C. Second, you can't return a procedure or function as a value, or assign a procedure or function to a global variable (procedures or function values aren't first class citizens). The reason for this restriction is that procedure-valued variables need to be closures -- that is, they need to include a reference to the procedure body along with a pointer to the environment of definition of the procedure. With a stack-based storage allocation scheme, if we tried to assign a procedure to a global variable, environment of definition may have disappeared when we try to call the procedure.

**HEAP-BASED STORAGE Allocation**The most flexible allocation scheme is heap-based allocation. Here, storage can be allocated and deallocated dynamically at arbitrary times during program execution. This will be more expensive than either static or stack-based allocation. Heap-based allocation is used ubiquitously in languages such as Scheme and Smalltalk, and for objects in Java.

Issue: when is storage allocated and deallocated?

Allocation is easy. In C, malloc (a standard library function) allocates fresh storage. In Lisp/Scheme, a new cons cell is allocated when the cons function is called, array storage can be allocated using make-array, and so forth. In Java new storage is allocated when the program makes a new instance of a class.

Deallocation is harder. There are two approaches: programmer-controlled and automatic. In languages such as C, the programmer is in charge of deciding when heap storage can be freed (in C using the freefunction). This is efficient but can lead to problems if the programmer makes a mistake -- either storage is not freed even though it is no longer needed (memory leak), or is freed but referred to later (dangling pointer). Dangling pointers can lead to type insecurities or other errors -- one can refer to storage that has been re-allocated for another purpose with data of a different type. Some studies estimate that over 30% of development time on large C/C++ projects is related to storage management issues.

Java, Scheme, and various other languages, use automatic storage management. There is no explicit deallocate function; rather, storage is automatically reclaimed some time after it is no longer accessible. We'll look at some of the most common techniques for garbage collection below:

**ASYNCIO LIBRARY  
🡺 Asybcio Library  
*The Primitives***

asyncio is supposed to implement asynchronous IO with the help of coroutines. Originally implemented as a library around the yield and yield from expressions it's now a much more complex beast as the language evolved at the same time. So here is the current set of things that you need to know exist:  
event loops, event loop policies, awaitables, coroutine functions, old style coroutine functions, coroutines, coroutine wrappers, generators, futures, concurrent futures, tasks, handles, executors, transports, protocols

In addition the language gained a few special methods that are new:

\_\_aenter\_\_ and \_\_aexit\_\_ for asynchronous with blocks  
\_\_aiter\_\_ and \_\_anext\_\_ for asynchronous iterators (async loops and async comprehensions). For extra fun that protocol already changed once. In 3.5 it returns an awaitable (a coroutine) in Python 3.6 it will return a newfangled async generator.  
\_\_await\_\_ for custom awaitables

***Event Loops***

The event loop in asyncio is a bit different than you would expect from first look. On the surface it looks like each thread has one event loop but that's not really how it works. Here is how I think this works:

* if you are the main thread an event loop is created when you call asyncio.get\_event\_loop()
* if you are any other thread, a runtime error is raised from asyncio.get\_event\_loop()
* You can at any point asyncio.set\_event\_loop() to bind an event loop with the current thread. Such an event loop can be created with the asyncio.new\_event\_loop() function.
* Event loops can be used without being bound to the current thread.
* asyncio.get\_event\_loop() returns the thread bound event loop, it does not return the currently running event loop.

The combination of these behaviors is super confusing for a few reasons. First of all you need to know that these functions are delegates to the underlying event loop policy which is globally set. The default is to bind the event loop to the thread. Alternatively one could in theory bind the event loop to a greenlet or something similar if one would so desire. However it's important to know that library code does not control the policy and as such cannot reason that asyncio will scope to a thread.

Secondly asyncio does not require event loops to be bound to the context through the policy. An event loop can work just fine in isolation. However this is the first problem for library code as a coroutine or something similar does not know which event loop is responsible for scheduling it. This means that if you call asyncio.get\_event\_loop() from within a coroutine you might not get the event loop back that ran you. This is also the reason why all APIs take an optional explicit loop parameter. So for instance to figure out which coroutine is currently running one cannot invoke something like this:  
**def** get\_task**():**loop **=** asyncio**.**get\_event\_loop**()  
try  
return** asyncio**.**Task**.**get\_current**(**loop**)  
except** **RuntimeError:  
return** None

Instead the loop has to be passed explicitly. This furthermore requires you to pass through the loop explicitly everywhere in library code or very strange things will happen. Not sure what the thinking for that design is but if this is not being fixed (that for instance get\_event\_loop() returns the actually running loop) then the only other change that makes sense is to explicitly disallow explicit loop passing and require it to be bound to the current context (thread etc.).

Since the event loop policy does not provide an identifier for the current context it also is impossible for a library to "key" to the current context in any way. There are also no callbacks that would permit to hook the tearing down of such a context which further limits what can be done realistically.  
  
***Awaitables and Coroutines***  
In my humble opinion the biggest design mistake of Python was to overload iterators so much. They are now being used not just for iteration but also for various types of coroutines. One of the biggest design mistakes of iterators in Python is that StopIteration bubbles if not caught. This can cause very frustrating problems where an exception somewhere can cause a generator or coroutine elsewhere to abort. This is a long running issue that Jinja for instance has to fight with. The template engine internally renders into a generator and when a template for some reason raises a StopIteration the rendering just ends there.

Python is slowly learning the lesson of overloading this system more. First of all in 3.something the asyncio module landed and did not have language support. So it was decorators and generators all the way down. To implemented the yield from support and more, the StopIterationwas overloaded once more. This lead to surprising behavior like this:

>>> **def** foo**(**n**):**  
 **if** n **in** **(0,** **1):**  
... **return** **[1]**  
... **for** item **in** range**(**n**):**  
... **yield** item **\*** **2**  
>>> list**(**foo**(0))**  
*[]*  
>>> list**(**foo**(1))**  
*[]*  
>>> list**(**foo**(2))**  
*[0, 2]*

No error, no warning. Just not the behavior you expect. This is because a return with a value from a function that is a generator actually raises a StopIteration with a single arg that is not picked up by the iterator protocol but just handled in the coroutine code.  
With 3.5 and 3.6 a lot changed because now in addition to generators we have coroutine objects. Instead of making a coroutine by wrapping a generator there is no a separate object which creates a coroutine directly. It's implemented by prefixing a function with async. For instance async def x()will make such a coroutine. Now in 3.6 there will be separate async generators that will raise AsyncStopIteration to keep it apart. Additionally with Python 3.5 and later there is now a future import (generator\_stop) that will raise a RuntimeError if code raises StopIteration in an iteration step.  
Why am I mentioning all this? Because the old stuff does not really go away. Generators still have send and throw and coroutines still largely behave like generators. That is a lot of stuff you need to know now for quite some time going forward.  
To unify a lot of this duplication we have a few more concepts in Python now:

* awaitable: an object with an \_\_await\_\_ method. This is for instance implemented by native coroutines and old style coroutines and some others.
* coroutinefunction: a function that returns a native coroutine. Not to be confused with a function returning a coroutine.
* a coroutine: a native coroutine. Note that old asyncio coroutines are not considered coroutines by the current documentation as far as I can tell. At the very least inspect.iscoroutine does not consider that a coroutine. It's however picked up by the future/awaitable branches.

In particularly confusing is that asyncio.iscoroutinefunction and inspect.iscoroutinefunction are doing different things. Same with inspect.iscoroutine and inspect.iscoroutinefunction. Note that even though inspect does not know anything about asycnio legacy coroutine functions in the type check, it is apparently aware of them when you check for awaitable status even though it does not conform to \_\_await\_\_.  
***Coroutine Wrappers***

Whenever you run async def Python invokes a thread local coroutine wrapper. It's set with sys.set\_coroutine\_wrapper and it's a function that can wrap this. Looks a bit like this:  
>>> import sys  
>>> sys.set\_coroutine\_wrapper(lambda x: 42)  
>>> async def foo():  
... pass  
>>> foo()  
\_\_main\_\_:1: RuntimeWarning: coroutine 'foo' was never awaited  
42

In this case I never actually invoke the original function and just give you a hint of what this can do. As far as I can tell this is always thread local so if you swap out the event loop policy you need to figure out separately how to make this coroutine wrapper sync up with the same context if that's something you want to do. New threads spawned will not inherit that flag from the parent thread.

This is not to be confused with the asyncio coroutine wrapping code.  
  
***Awaitables and Futures***Some things are awaitables. As far as I can see the following things are considered awaitable:

* native coroutines
* generators that have the fake CO\_ITERABLE\_COROUTINE flag set (we will cover that)
* objects with an \_\_await\_\_ method  
  Essentially these are all objects with an \_\_await\_\_ method except that the generators don't for legacy reasons. Where does the CO\_ITERABLE\_COROUTINE flag come from? It comes from a coroutine wrapper (now to be confused with sys.set\_coroutine\_wrapper) that is @asyncio.coroutine. That through some indirection will wrap the generator with types.coroutine (to to be confused withtypes.CoroutineType or asyncio.coroutine) which will re-create the internal code object with the additional flag CO\_ITERABLE\_COROUTINE.  
  So now that we know what those things are, what are futures? First we need to clear up one thing: there are actually two (completely incompatible) types of futures in Python 3. asyncio.futures.Futureand concurrent.futures.Future. One came before the other but they are also also both still used even within asyncio. For instance asyncio.run\_coroutine\_threadsafe() will dispatch a coroutine to a event loop running in another thread but it will then return a concurrent.futures.Future object instead of aasyncio.futures.Future object. This makes sense because only the concurrent.futures.Future object is thread safe.  
  So now that we know there are two incompatible futures we should clarify what futures are in asyncio. Honestly I'm not entirely sure where the differences are but I'm going to call this "eventual" for the moment. It's an object that eventually will hold a value and you can do some handling with that eventual result while it's still computing. Some variations of this are called deferreds, others are called promises. What the exact difference is is above my head.  
  What can you do with a future? You can attach a callback that will be invoked once it's ready or you can attach a callback that will be invoked if the future fails. Additionally you can await it (it implements \_\_await\_\_ and is thus awaitable). Additionally futures can be cancelled.  
  So how do you get such a future? By calling asyncio.ensure\_future on an awaitable object. This will also make a good old generator into such a future. However if you read the docs you will read thatasyncio.ensure\_future actually returns a Task. So what's a task?  
    
  ***Tasks***  
  A task is a future that is wrapping a coroutine in particular. It works like a future but it also has some extra methods to extract the current stack of the contained coroutine. We already saw the tasks mentioned earlier because it's the main way to figure out what an event loop is currently doing via Task.get\_current.  
  There is also a difference in how cancellation works for tasks and futures but that's beyond the scope of this. Cancellation is its own entire beast. If you are in a coroutine and you know you are currently running you can get your own task through Task.get\_current as mentioned but this requires knowledge of what event loop you are dispatched on which might or might not be the thread bound one.  
  It's not possible for a coroutine to know which loop goes with it. Also the Task does not provide that information through a public API. However if you did manage to get hold of a task you can currently access task.\_loop to find back to the event loop.  
   ***Handles***In addition to all of this there are handles. Handles are opaque objects of pending executions that cannot be awaited but they can be cancelled. In particular if you schedule the execution of a call with call\_soon or call\_soon\_threadsafe (and some others) you get that handle you can then use to cancel the execution as a best effort attempt but you can't wait for the call to actually take place.  
    
  ***Executors***Since you can have multiple event loops but it's not obvious what the use of more than one of those things per thread is the obvious assumption can be made that a common setup is to have N threads with an event loop each. So how do you inform another event loop about doing some work? You cannot schedule a callback into an event loop in another thread and get the result back. For that you need to use executors instead.Executors come from concurrent.futures for instance and they allow you to schedule work into threads that itself is not evented. For instance if you use run\_in\_executor on the event loop to schedule a function to be called in another thread. The result is then returned as an asyncio coroutine instead of a concurrent coroutine like run\_coroutine\_threadsafe would do. I did not yet have enough mental capacity to figure out why those APIs exist, how you are supposed to use and when which one. The documentation suggests that the executor stuff could be used to build multiprocess things.  
    
  ***Transports and Protocols***  
  I always though those would be the confusing things but that's basically a verbatim copy of the same concepts in Twisted. So read those docs if you want to understand them.  
  ***How to use asyncio***  
  Now that we know roughly understand asyncio I found a few patterns that people seem to use when they write asyncio code:
* pass the event loop to all coroutines. That appears to be what a part of the community is doing. Giving a coroutine knowledge about what loop is going to schedule it makes it possible for the coroutine to learn about its task.
* alternatively you require that the loop is bound to the thread. That also lets a coroutine learn about that. Ideally support both. Sadly the community is already torn of what to do.
* If you want to use contextual data (think thread locals) you are a bit out of luck currently. The most popular workaround is apparently atlassian's aiolocals which basically requires you to manually propagate contextual information into coroutines spawned since the interpreter does not provide support for this. This means that if you have a utility library spawning coroutines you will lose context.
* Ignore that the old coroutine stuff in Python exists. Use 3.5 only with the new async defkeyword and co. In particular you will need that anyways to somewhat enjoy the experience because with older versions you do not have async context managers which turn out to be very necessary for resource management.
* Learn to restart the event loop for cleanup. This is something that took me longer to realize than I wish it did but the sanest way to deal with cleanup logic that is written in async code is to restart the event loop a few times until nothing pending is left. Since sadly there is no common pattern to deal with this you will end up with some ugly workaround at time. For instance aiohttp's web support also does this pattern so if you want to combine two cleanup logics you will probably have to reimplement the utility helper that it provides since that helper completely tears down the loop when it's done. This is also not the first library I saw do this :(
* Working with subprocesses is non obvious. You need to have an event loop running in the main thread which I suppose is listening in on signal events and then dispatches it to other event loops. This requires that the loop is notified viaasyncio.get\_child\_watcher().attach\_loop(...).
* Writing code that supports both async and sync is somewhat of a lost cause. It also gets dangerous quickly when you start being clever and try to support with and async with on the same object for instance.
* If you want to give a coroutine a better name to figure out why it was not being awaited, setting \_\_name\_\_ doesn't help. You need to set \_\_qualname\_\_ instead which is what the error message printer uses.
* Sometimes internal type conversations can screw you over. In particular the asyncio.wait()function will make sure all things passed are futures which means that if you pass coroutines instead you will have a hard time finding out if your coroutine finished or is pending since the input objects no longer match the output objects. In that case the only real sane thing to do is to ensure that everything is a future upfront.  
    
  ***Context Data***Aside from the insane complexity and lack of understanding on my part of how to best write APIs for it my biggest issue is the complete lack of consideration for context local data. This is something that the node community learned by now. continuation-local-storage exists but has been accepted as implemented too late. Continuation local storage and similar concepts are regularly used to enforce security policies in a concurrent environment and corruption of that information can cause severe security issues.  
  The fact that Python does not even have any store at all for this is more than disappointing. I was looking into this in particular because I'm investigating how to best support [Sentry's breadcrumbs](https://docs.sentry.io/learn/breadcrumbs/) for asyncio and I do not see a sane way to do it. There is no concept of context in asyncio, there is no way to figure out which event loop you are working with from generic code and without monkeypatching the world this information will not be available.  
  Node is currently going through the process of [finding a long term solution for this problem](https://github.com/nodejs/node-eps/pull/18). That this is not something to be left ignored can be seen by this being a recurring issue in all ecosystems. It comes up with JavaScript, Python and the .NET environment. The problem [is named async context propagation](https://docs.google.com/document/d/1tlQ0R6wQFGqCS5KeIw0ddoLbaSYx6aU7vyXOkv-wvlM/edit) and solutions go by many names. In Go the context package needs to be used and explicitly passed to all goroutines (not a perfect solution but at least one). .NET has the best solution in the form of local call contexts. It can be a thread context, an web request context, or something similar but it's automatically propagating unless suppressed. This is the gold standard of what to aim for. Microsoft had this solved since more than 15 years now I believe.  
  I don't know if the ecosystem is still young enough that logical call contexts can be added but now might still be the time.