301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

AP-28*: Python: OOP, iterators and the GIL*

Next topics

- OO Programming in Python, multiple inheritance
- Iterators and generators
- Garbage Collection, multi-threading and the GIL

OOP in Python

Typical ingredients of the Object Oriented Paradigm:

- **Encapsulation**: dividing the code into a public **interface**, and a private **implementation** of that interface;
- **Inheritance**: the ability to create **subclasses** that contain specializations of their parent classes.
- **Polymorphism**: The ability to **override** methods of a Class by extending it with a subclass (inheritance) with a more specific implementation (**inclusion polymorphism**)

From https://docs.python.org/3/tutorial/classes.html:

"Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows ***multiple base classes****, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and* ***can be modified further after creation****."*

Defining a class (object)

- A class is a blueprint for a new data type with specific internal ***attributes*** (like a struct in C) and internal functions (***methods***).
- To declare a class in Python the syntax is the following:

```
class className:
    <statement-1>
    …
    <statement-n>
```

- **`statements`** are assignments or function definitions
- A new namespace is created, where all names introduced in the statements will go.
- When the class definition is left, a ***class object*** is created, bound to **className**, on which two operations are defined: ***attribute reference*** and ***class instantiation***.
- *Attribute reference* allows to access the names in the namespace in the usual way

Example: Attribute reference on a class object

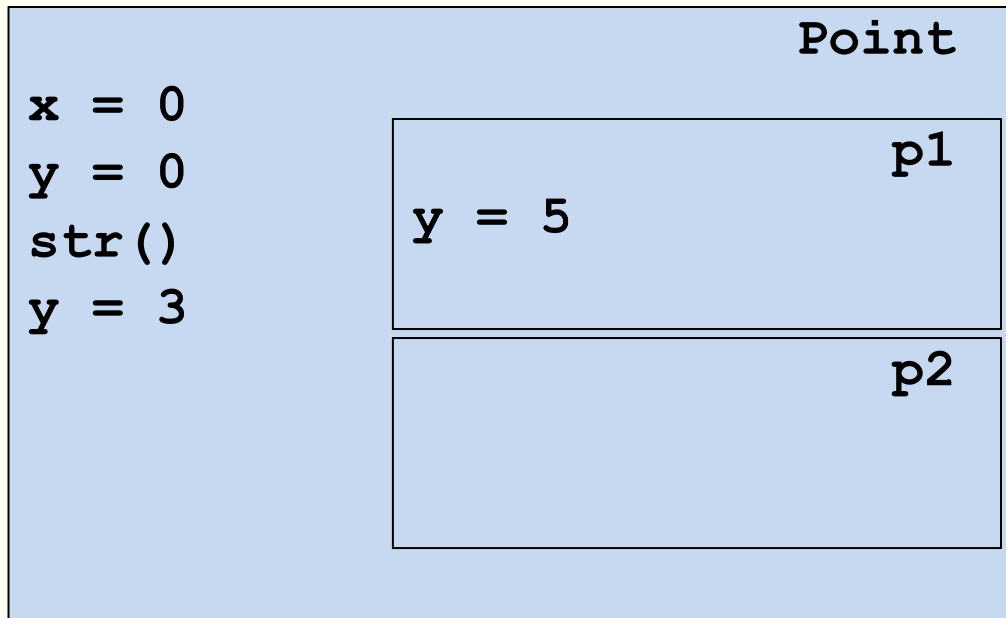
```
class Point:
  x = 0
  y = 0
  def str(): # no capture: needs qualified names to refer to x and y
      return  "x = " + (str) (Point.x) + ", y = " + (str) (Point.y)
#--------
import ...
>>> Point.x
0
>>> Point.y = 3
>>> Point.z = 5 # adding new name
>>> Point.z
5
>>> def add(m,n):
      return m+n
>>> Point.sum = add # adding new function
>>> Point.sum(3,4)
7
```

```
                    Point
x = 0
y = 0
str()
y = 3
z = 5
sum = add(m,n)
```

Creating a class instance

- A **class instance** introduces a **new namespace** ***nested in the class namespace***: by visibility rules all names of the class are visible
- If no *constructor* is present, the syntax of class instantiation is **className():** the new namespace is empty

```
class Point:
  x = 0
  y = 0
  def str():
      return  "x = " + str(Point.x) + ", y = " + str(Point.y)
#--------
>>> p1 = Point()
>>> p2 = Point()
>>> p1.x
0
>>> Point.y = 3
>>> p2.y
3
>>> p1.y = 5
>>> p2.y
3
```



Instance methods

- A class can define a set of ***instance methods***, which are just functions:

```
def methodname(self, parameter_1, ..., parameter_n):
        statements
```

- The first argument, usually called self, represents the ***implicit parameter*** (**`this`** in Java)

- A method ***must*** access the object's attributes through the **`self`** reference (eg. **self.x**) and the class attributes using **className.<attrName>** (or **self.__class__.<attrName>**)

- The first parameter must not be passed when the method is called with dot-notation on an object. It is bound to the target object. Syntax:

```
obj.methodname(arg_1, ..., arg_n):
```

- But it can be passed explicitly. Alternative syntax:

```
className.methodname(obj, arg_1, ..., arg_n):
```

"Instance methods"

- Any function ***with at least one parameter*** defined in a class can be invoked on an instance of the class with the dot notation.

```
class Foo
    def fun(par-0, par-1, ..., par-n):
          statements
#----
>>> obj = Foo()
>>> obj.fun(arg-1,...,arg-n)
# is syntactic sugar for
>>> obj.__class__.fun(obj,arg-1,...,arg-n)
```

- Since the instance **`obj`** is bound to the first parameter, **`par-0`** is usually called **`self`**.
- A name **`x`** defined in the (namespace of the) instance is accessed as **`par-0.x`** (i.e., usually **`self.x`**)
- A name **`x`** defined in the class is accessed as **`className.x`** (or **`self.__class__.x`**)

Constructors

- A constructor is a **special instance method** with name `__init__`.

Syntax:

```
def __init__(self, parameter1, ..., parametern):
        statements
```

- Invocation: `obj = className(arg1, …, argn)`
- The first parameter `self` is bound to the new object.
- `statements` typically initialize (thus create) "instance variables", i.e. names in the new object namespace.
- Note: at most ONE constructor (**no overloading in Python**!)

```
class Point:
   instances = []
   def __init__(self, x=0, y=0):
      self.x = x
      self.y = y
      Point.instances.append(self)
#--------
>>> p1 = Point(3,4)
```

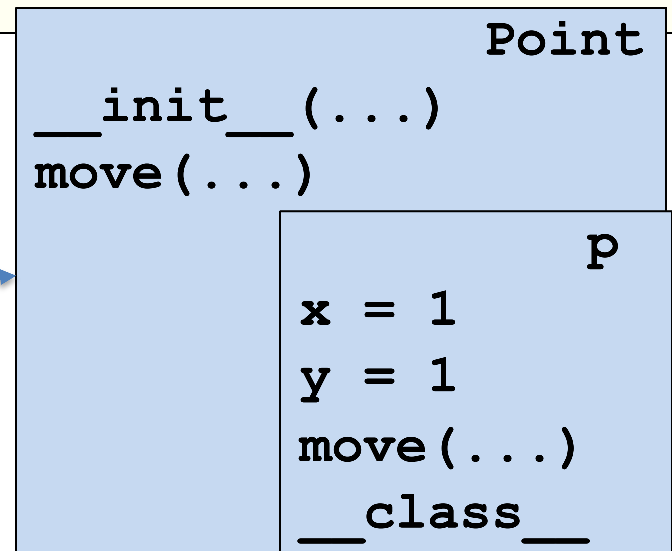
```
                             Point
instances = [<Point
object at ...>]

                    p1
        x = 3
        y = 4
```

What about "methods in instances?"

- Instances are themselves namespaces: we can add functions to them.
- Applying the usual rules, they can hide "instance methods"

```
class Point:
   def __init__(self, x, y):
      self.x = x
      self.y = y
      def move(z,t):
         self.x -= z
         self.y -= t
      self.move = move
   def move(self,dx,dy):
      self.x += dx
      self.y += dy
```

```
>>> p = Point(1,1)
>>> p.x
1
>>> p.move(1,1)
>>> p.x
0
>>> p.__class__.move(p,2,2)
>>> p.x
2
```



String representation

- It is often useful to have a textual representation of an object with the values of its attributes. This is possible with the following instance method:

```
def __str__(self) :
        return <string>
```

- This is equivalent to Java's **toString** (converts object to a string) and it is invoked automatically when **str** or **print** is called.

Special methods

- **Method overloading**: you can define special instance methods so that Python's built-in operators can be used with your class

Binary Operators

Operator	Class Method
-	`__sub__(self, other)`
+	`__add__(self, other)`
*	`__mul__(self, other)`
/	`__truediv__(self, other)`

Operator	Class Method
==	`__eq__(self, other)`
!=	`__ne__(self, other)`
<	`__lt__(self, other)`
>	`__gt__(self, other)`
<=	`__le__(self, other)`
>=	`__ge__(self, other)`

Unary Operators

Operator	Class Method
-	`__neg__(self)`
+	`__pos__(self)`

```
class Point: # example
  ...
  def __add__(self,other):
      return Point(self.x + other.x,
                   self.y + other.y)
  def __neg__(self):
      return Point(-self.x, - self.y)
```

- Analogous to C++ overloading mechanism:
  - **Pros**: very compact syntax
  - **Cons**: may be more difficult to read if not used with care

(Multiple) Inheritance, in one slide

- A class can be defined as a *derived class*

```
class derived(baseClass):
    statements
    statements
```

- No need of additional mechanisms: the namespace of **derived** is nested in the namespace of **baseClass**, and uses it as the next non-local scope to resolve names
- All instance methods are automatically virtual: lookup starts from the instance (namespace) where they are invoked
- Python supports **multiple inheritance**

```
class derived(base1,..., basen):
    statements
    statements
```

- ***Diamond problem*** solved by an algorithm that linearizes the set of all (directly or indirectly) inherited classes: the **Method resolution order** (**MRO**), using the **C3** algorithm ➔ `ClassName.mro()`
- https://www.python.org/download/releases/2.3/mro/

Encapsulation (and "name mangling")

- **Private** instance variables (not accessible except from inside an object) **don't exist in Python.**

- **Convention:** a **name prefixed with underscore** (e.g. `_spam`) is treated as ***non-public part of the API*** (function, method or data member). It should be considered an implementation detail and subject to change without notice.

Name mangling ("storpiatura")

- Sometimes class-private members are needed to avoid clashes with names defined by subclasses. Limited support for such a mechanism, called *name mangling.*

- Any **name with at least two leading underscores and at most one trailing underscore** like e.g. `__spam` is textually replaced with `_Class__spam`, where `Class` is the current class name.

Uses of Name Mangling

- **Avoiding Name Clashes**: When designing a class hierarchy, you might define attributes that are intended to be used only within a specific class. Name mangling helps avoid accidental name clashes when a subclass defines an attribute with the same name.
- **Implementing Encapsulation**: While Python does not have private variables in the strict sense, name mangling provides a way to make attributes ***less accessible*** from outside the class, thus enforcing encapsulation to some extent.
- **Frameworks and Libraries**: When developing frameworks or libraries, you might use name mangling to avoid conflicts with attributes defined by the users of your framework or library.

Name mangling to avoid name clashes

```
class BaseClass:
    def __init__(self):
        self.__mangled_attr = "BaseClass attribute"

    def get_mangled_attr(self):
        return self.__mangled_attr

class SubClass(BaseClass):
    def __init__(self):
        super().__init__()
        self.__mangled_attr = "SubClass attribute"

    def get_subclass_attr(self):
        return self.__mangled_attr

base_obj = BaseClass()
sub_obj = SubClass()

print(base_obj.get_mangled_attr())  # “BaseClass attribute”
print(sub_obj.get_mangled_attr())   # “BaseClass attribute”
print(sub_obj.get_subclass_attr())  # “SubClass attribute”
```

Name mangling to avoid breaking logic

- Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls.

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.update(iterable)      # comment this
#       self.__update(iterable)    # uncomment this
    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

#   __update = update # copy of update(): uncomment

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Limitations

- **Name mangling is not foolproof**: While it makes attribute names harder to access, it is still possible to access them if one knows the mangled name. For instance, `_ClassName__attr` can be used to access the attribute directly.
- **Readability**: Overuse of name mangling can make the code harder to read and understand. It should be used judiciously to balance between avoiding name clashes and maintaining code readability.
- **Dynamic Class Names**: If you use dynamic class names (e.g., using **type()** to create classes), name mangling can become less predictable and harder to manage.

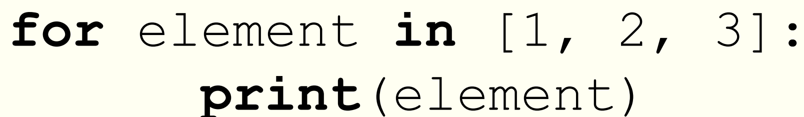
Static methods and class methods

- **Static methods** are simple functions defined in a class with no `self` argument, preceded by the **@staticmethod** decorator
- They are defined inside a class but they cannot access instance attributes and methods
- They can be called through both the class and any instance of that class!
- They allow subclasses to customize the static methods with inheritance. Classes can inherit static methods without redefining them.

- Class methods are similar to static methods but they have a first parameter which is the class name.
- Definition must be preceded by the **@classmethod** decorator
- Can be invoked on the class or on an instance.

Iterators

- An **iterator** is an object which allows a programmer to traverse through all the elements of a collection (**iterable** object), regardless of its specific implementation. In Python they are used implicitly by the **FOR** loop construct.
- **Iterable** objects must support method `__iter__()`, returing the iterator
- **Iterators** must support methods:
  - `__iter__()` returning the iterator object itself
  - `__next__()` returning the next value. It raises a `StopIteration` exception if there are no more items to return
- An iterator object can be used only once. After it raises `StopIteration` once, it will keep raising the same exception.
- Example:




```
>>> list = [1,2,3]
>>> it = iter(list)
>>> it
<listiterator object at 0x00A1DB50>
>>> it.__next__()
1
>>> it.__next__()
2
>>> it.__next__()
3
>>> it.__next__() -> raises StopIteration
```

Generators and coroutines

- **Generators** are a simple and powerful tool for creating iterators.
- They are written like **regular functions** but use the **`yield`** statement whenever they want to return data.
- Each time the **`next()`** is called, the generator resumes where it left-off (it remembers all the data values and which statement was last executed).
- ***Anything that can be done with generators can also be done with class based iterators (not vice-versa).***
- What makes generators so compact is that the **`__iter__()`** and **`next()`** methods are created automatically.
- Another key feature is that the local variables and execution state are **automatically saved** between calls.

Generators (2)

- In addition to automatic method creation and saving program state, when generators terminate, they automatically raise **`StopIteration`**.
- In combination, these features make it easy to create iterators with no more effort than writing a regular function.

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

#----------------

>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Typing in Python

- Dynamic, strong duck typing
- Code can be annotated with types

```
def greetings(name: str) -> str:
    return 'Hello ' + name.
```

- Module **typing** provides runtime support for type hints
- Type hints can be checked statically by external tools, like **mypy**
- They are ignored by CPython

On Polymorphism in Python

- **Overloading**: forbidden, but its absence alleviated by:
  - Default parameters for functions
  - Dynamic typing
  - Duck typing
- **Overriding**: ok, thanks to nesting of namespaces
- **Generics**: type hints (module **typing** + **mypy** support generics)

Garbage collection in Python

CPython manages memory with a **reference counting** + a **mark&sweep** cycle collector scheme

- Reference counting: each object has a counter storing the number of references to it. When it becomes 0, memory can be reclaimed.
- Pros: simple implementation, memory is reclaimed as soon as possible, no need to freeze execution passing control to a garbage collector
- Cons: additional memory needed for each object; cyclic structures in garbage cannot be identified (thus the need of mark&sweep)

Memory safety in Python

Getting the reference count:

```
import ctypes

my_list = [1, 2, 3]

# finding the id of list object
my_list_address = id(my_list)

# finds reference count of my_list
ref_count = ctypes.c_long.from_address(my_list_address).value

print(f"Ref count for my_list is: {ref_count}")
```

No explicit deallocation on the heap

- **del** removes entries from the namespace

Therefore:

- No **dangling pointers** in Python
- No **double free** in Python

Race conditions in Python? NO

Example: Shared counter incremented 10k times in parallel by two threads.

```
# counter in closure
def counter_factory():
  counter = 0
  def counter_increaser():
      nonlocal counter
      counter = counter + 1
  return counter_increaser
```

```
# decorator: repeats fun ntimes
def times(ntimes):
 """Usage:
times(ntimes)(fun)(args,kwargs)"""
    def times_dec(fun):
        def wrapper(*args,**kwargs):
            for i in range(ntimes):
                fun(*args,**kwargs)
            return
        return wrapper
    return times_dec
```

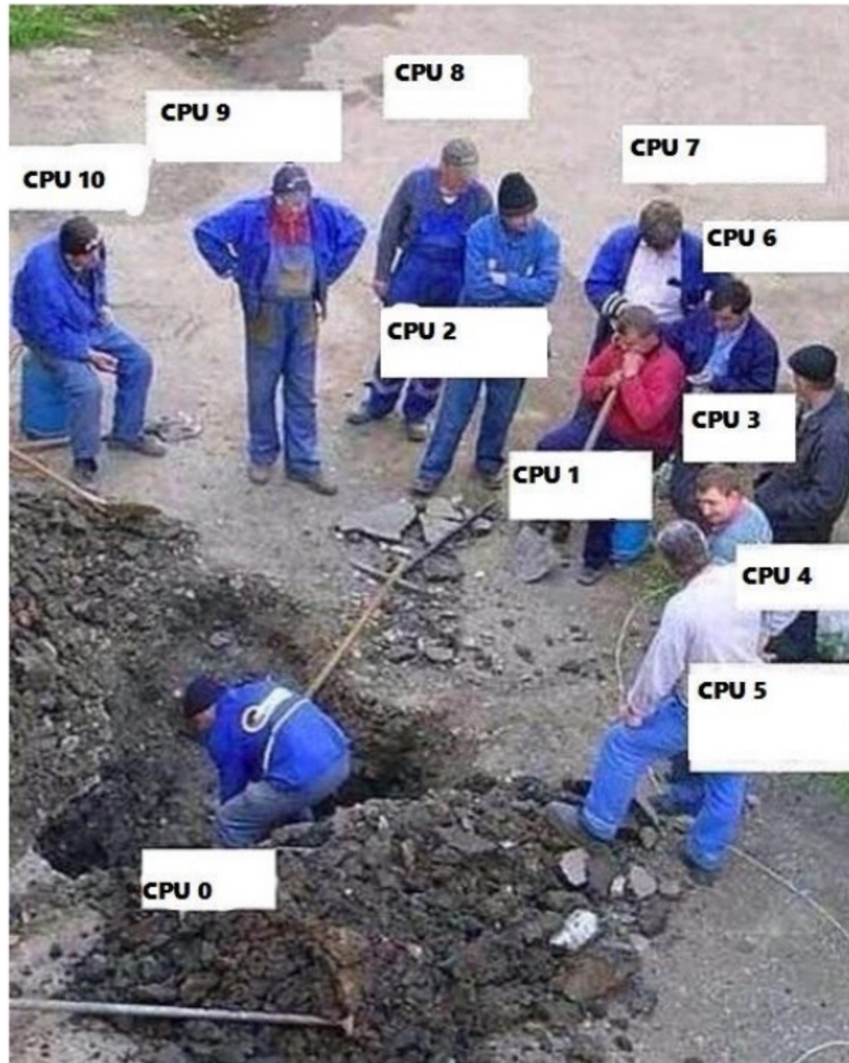
```
# Runs fun() in parallel
def thread_fun(nthreads, fun):
    threads = []
    for _ in range(nthreads):
        threads.append(Thread(target = fun))
        threads[-1].start()
    for t in threads:
        t.join()
```

```
inc = counter_factory()
thread_fun(2,times(10000)(inc))
inc.__closure__[0].cell_contents
```

Handling reference counters

- Updating the refcount of an object has to be done atomically
- In case of multi-threading you need to synchronize all the times you modify refcounts, or else you can have wrong values
- Synchronization primitives are quite expensive on contemporary hardware
- Since almost every operation in CPython can cause a refcount to change somewhere, handling refcounts with some kind of synchronization would cause *spending almost all the time on synchronization*
- As a consequence…

Concurrency in Python…



The Global Interpreter Lock (GIL)

- The CPython interpreter assures that only one native thread executes Python bytecodes at a time, thanks to the **Global Interpreter Lock**, which is a mutex on the Python interpreter
- The current thread must hold the **GIL** before it can safely access Python objects
- This simplifies the CPython implementation by making the object model (including critical built-in types such as **dict**) implicitly safe against concurrent access: no race conditions
- Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, **at the expense of much of the parallelism afforded by multi-processor machines**.

More on the GIL

- However the GIL can degrade performance even when it is not a bottleneck. The system call overhead is significant, especially on multicore hardware.
- Two threads calling a function may take twice as much time as a single thread calling the function twice.
- The GIL can cause I/O-bound threads to be scheduled ahead of CPU-bound threads. And it prevents signals from being delivered.
- Some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing.
- Also, the GIL is always released when doing I/O.

Alternatives to the GIL?

- Past efforts to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case.
- It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.
- Guido van Rossum has said he will reject any proposal in this direction that slows down single-threaded programs.
- **Jython** (on JVM, -> 2017, Python 2.7) and **IronPython** (on .NET) have no GIL and can fully exploit multiprocessor systems
- **PyPy** (Python in Python, supporting JIT) currently has a GIL like CPython
- in **Cython** (compiled, for CPython extension modules) the GIL exists, but can be released temporarily using a "with" statement

Criticisms to Python: syntax of tuples

```
>>> type((1,2,3))
<class 'tuple'>
>>> type(())
<class 'tuple'>
>>> type((1))
<class 'int'>
>>> type((1,))
<class 'tuple'>
```

- Tuples are made by the commas, not by ( )
- With the exception of the empty tuple…

Experimental feature:
GIL optional in Python 3.13 (Oct. 2024)

- **Experimental Support for Free-Threaded Mode** where the GIL is disabled. This is aimed at improving multi-threading capabilities and enabling better performance in CPU-bound tasks.
- **Specializing Interpreter Enhancements**: The specializing interpreter (interpreter with some ad hoc optimizations) has undergone modifications to ensure thread safety without the GIL.
- **New Py_mod_gil Slot:** Extensions can now define a new PEP 489-style **Py_mod_gil** slot to manage GIL behavior when loading modules. If this slot is not properly set, the interpreter will enable the GIL and pause all threads, providing warnings to users.
- **PYTHONGIL Environment Variable**: Users can control GIL behavior at runtime using the PYTHONGIL environment variable. Setting it to 0 forces the GIL to remain disabled, while setting it to 1 forces it to be enabled.
- **Non-Generational Garbage Collection**: The GIL changes support a shift from generational cyclic garbage collection to a non-generational model, aimed at reducing thread pauses during garbage collection cycles and improving multi-threading efficiency.
- The feature is potentially reversible if it breaks more of the current implementation of Cpython than expected.

Criticisms to Python: indentation

- Lack of brackets makes the syntax "weaker" than in other languages: accidental changes of indentation may change the semantics, leaving the program syntactically correct.

```
def foo(x):
    if x == 0:
        bar()
        baz()
    else:
        qux(x)
        foo(x - 1)
```

```
def foo(x):
    if x == 0:
        bar()
        baz()
    else:
        qux(x)
    foo(x - 1)
```

- Mixed use of tabs and blanks may cause bugs almost impossible to detect

Criticisms to Python: indentation

- Lack of brackets makes it harder to refactor the code or insert new one
- "When I want to refactor a bulk of code in Python, I need to be very careful. Because if lost, I'm not sure what I'm editing belongs to which part of the code. Python depends on indentation, so if I have mistakenly removed some indentation, I totally have no idea whether the correct code should belong to that **if** clause or this **while** clause."
- Will Python change in the future?

```
>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
>>>
```

Builtins & Libraries

- The Python ecosystem is extremely rich and in fast evolution
- For available functions, classes and modules browse:
  - **Builtin Functions**
    - https://docs.python.org/3.13/library/functions.html
  - **Standard library**
    - https://docs.python.org/3.13/tutorial/stdlib.html
- There are dozens of other libraries, mainly for scientific computing, machine learning, computational biology, data manipulation and analysis, natural language processing, statistics, symbolic computation, etc.

Python libraries…

Top 30 Python Libraries List

Rank	Library	Primary Use Case
1	NumPy	Scientific Computing
2	Pandas	Data Analysis
3	Matplotlib	Data Visualization
4	SciPy	Scientific Computing
5	Scikit-learn	Machine Learning
6	TensorFlow	Machine Learning/AI
7	Keras	Machine Learning/AI
8	PyTorch	Machine Learning/AI
9	Flask	Web Development
10	Django	Web Development
11	Requests	HTTP for Humans
12	BeautifulSoup	Web Scraping
13	Selenium	Web Testing/Automation
14	PyGame	Game Development
15	SymPy	Symbolic Mathematics
16	Pillow	Image Processing
17	SQLAlchemy	Database Access
18	Plotly	Interactive Visualization
19	Dash	Web Applications
20	Jupyter	Interactive Computing
21	FastAPI	Web APIs
22	PySpark	Big Data Processing
23	NLTK	Natural Language Processing
24	spaCy	Natural Language Processing
25	Tornado	Web Development
26	Streamlit	Data Apps
27	Bokeh	Data Visualization
28	PyTest	Testing Framework
29	Celery	Task Queuing
30	Gunicorn	WSGI HTTP Server