I ADVANCED OR RECENTLY ADDED ELEMENTS OF SYNTAX

I.1 Partial functions

- It is possible, for a long time now, to define a function that returns a function.
- This helps in implementing something like this:

```
def get_is_small(limit):
    def is_small(i):
        return i < limit
    return is_small
```

- The result is that you have a function reusable in many different situations, without having to keep track of the *`limit`* value:

```
>>> is_quite_small = get_is_small(15)
>>> is_very_small = get_is_small(3)
>>> is_quite_small(6)
True
>>> is_very_small(6)
False
```

- Python 2.5 and higher offer a general purpose function (**`partial()`**) to provide the outer function for you. This function belongs to the module **`functools`**.
- Now you can write:

```
from functools import partial

def is_small(limit, i):
    return i < limit

is_quite_small = partial(is_small, 15)
is_very_small  = partial(is_small,3)
```

I.2 Iterators

- You have probably noticed that most container objects can be looped over using a `for` statement. The `for` loop works with all object that can be considered as ***iterators***:

```
for element in [1, 2, 3]:
     print(element)

for line in open("myfile.txt"):
     print(line)
```

- This style of access is clear, concise, and convenient.

- Behind the scenes, the `for` statement calls **`iter()`** on the container object. The function returns an iterator object that defines the method `next()` which accesses elements in the container one at a time.

- When there are no more elements, `next()` raises a `StopIteration` exception which tells the for loop to terminate.

- To add an iterator to a class, you only have to define a **`__iter__()`** method which returns an object with a `__next__()` method. If the class defines `__next__()`, then `__iter__()` can just return self:

```
class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

for char in Reverse('spam'):
    print(char,sep='')
print()
for i in Reverse([1,3,5,7,9]):
    print(i,sep='')
print()
```

I.3 Generators

I.3.1 Overview

- When the body of a function contains one or more occurrences of the keyword **`yield`**, the function is called a ***generator***.

- When a *generator* is called, the function body does not execute. Instead, calling the generator returns a special **`iterator`** object that wraps the function body, the set of its local variables (including its parameters), and the current point of execution, which is initially the start of the function.

- When the **`next()`** method of this `iterator` object is called, the function body executes up to the next `yield` statement, which takes the form:

  `yield` *`expression`*

- When a yield statement executes, the function is frozen with its execution state and local variables intact, and the expression following `yield` is returned as the result of the `next()` method.
  On the next call to `next()`,execution of the function body resumes where it left off, again up to the next `yield` statement.

- If the function body ends or executes a `return` statement, the `iterator` raises a `StopException` to indicate that the `iterator` is finished.

Note: `return` statements in a generator cannot contain expressions.

- The most common way to use an `iterator` is to loop on it with a `for` statement, you typically call a generator like this:

  `for` *`avariable`* `in` *`somegenerator`*`(`*`arguments`*`):`

- Here is a generator that works somewhat like the built-in `xrange()` function, but returns a sequence of floating-point values instead of a sequence of integers:

```
def frange(start, stop, step=1.0):
    while start < stop:
        yield start
        start += step
for i in frange (2.3, 5.67):
    print(i)
```

```
2.3
3.3
4.3
5.3
```

I.3.2 Generator as co-routines

- In languages that provide for generators, an important feature is the ability to pass a value back into the generator. This allows for supporting a programming feature called ***coroutines***.

- In order to make the generators more powerful, the designers of Python have added in Python 2.5 the ability to pass data back into the generator.

- If, from the perspective of the generator, you think of the `yield` as calling something, then the concept is easy: You just save the results of `yield`:

  ```
  x = yield start
  ```

- From the perspective of the caller of the generator, this statement returns control back to the caller, just as before. From the perspective of the generator, when the execution comes back into the generator, a value will come with it (in this case, the generator saves it into the variable x).

- Where does the value come from? The caller calls a function called `send()` to pass a value back into the generator. The function `send()`behaves just like the function `next()`, except that it passes a value.

- **`iterator`** objects also have a `close()`method (new in Python 2.5). This method frees up the resources for the generator.
  If you call `next()` again after calling `close()`, you'll get a `StopIteration` exception.

I.3.3 Generator Expressions

- Some simple generators can be coded succinctly as expressions using syntax similar to list comprehensions but with parentheses instead of square brackets.

- These expressions are designed for situations where the generator is used right away by an enclosing function or for loop.

- Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

- Instead of:

```
def power2(nb):
    for i in  range(1,nb+1):
        yield i*i

for val in power2(10):
    print(val)
```

- You can write:

```
for val in (i*i for i in range(1,11)):
    print(val)
```

- The utility of generator expressions is greatly enhanced when combined with reduction functions like `sum()`, `min()`, and `max()`.

- The following summation code will build a full list of squares in memory, iterate over those values, and, when the reference is no longer needed, delete the list:

```
sum([x*x for x in range(10)])
```

- Memory is conserved by using a generator expression instead:

```
sum(x*x for x in range(10))
```

- The syntax requires that a generator expression always needs to be directly inside a set of parentheses and cannot have a comma on either side.

- This means that you can write: `sum(x**2 for x in range(10))`
  but you would have to write:

```
import functools as fct
import operator as op
print(fct.reduce(op.add, (x**2 for x in range(10))))
```

I.4 Descriptors

I.4.1 Overview

- A descriptor is a way to customize what happens when you reference a class or instance attribute. Normally, Python just gets and sets values on attributes without any special processing. It's just basic storage. Sometimes, however, you might want to do more. You might need to validate the value that's being assigned to a value. You may want to retrieve a value and cache it for later use, so that future references don't have all the overhead.

- These are all things that would normally need to be done with a method, but if you've already started with a basic attribute, changing to a method would require changing all the code that uses the attribute to use a method call instead. This potential change is a primary motivation for typical Java programs to always use methods even for basic attribute access. The common pattern is to have all attributes private, and provide public access through methods, simply to accommodate potential future changes in the internals of that attribute access.

- Python's descriptors as an alternative approach. Instead of starting with methods all the time, you can start with basic attributes and write all the code you want. Then, if you ever need advanced processing to occur when you access those attributes, you can just add in a descriptor to do the work, without updating all the other code.

- A descriptor is a Python object that is assigned as an attribute of a class. The object is an instance of a class that provides special methods (described below), and the attribute it's assigned to is the one that will have the special processing. So the actual extra code will be inside the descriptor's class, rather than the class it will be assigned to.

- The methods a descriptor should provide are **`__get__()`**, **`__set__()`**, and **`__delete__()`**. If any of those methods are defined for an object, it is said to be a descriptor.

- The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, a.x has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the base classes of `type(a)` excluding metaclasses.

- If the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined.

Note: descriptors are only invoked for new style classes.

- Descriptors are a powerful, general purpose protocol. They are the mechanism behind properties, methods, static methods, class methods, and `super()`.

I.4.2 Descriptor Protocol

- A descriptor is implemented as a standard new-style class in Python, and it doesn't need to inherit from anything in particular besides object. The real trick to building a descriptor is defining at least one of the following three methods.

- Note that `obj` below returns to the object where the attribute was accessed, and `type` is the class where the descriptor was assigned as an attribute.

`__get__(self, obj, type=None)` *return a value*
`__set__(self, obj, value)` *returns None*
`__delete__(self, obj)` *returns None*

- If an object defines both `__get__()` and `__set__()`, it is considered a ***data descriptor***.

- Descriptors that only define `__get__()` are called ***non-data descriptors*** (they are typically used for methods but other uses are possible).

- Data and non-data descriptors differ in how overrides are calculated with respect to entries in an instance's dictionary:

    - If an instance's dictionary has an entry with the same name as a data descriptor, the data descriptor takes precedence.
    - If an instance's dictionary has an entry with the same name as a non-data descriptor, the dictionary entry takes precedence.

Note: to make a read-only data descriptor, define both `__get__()` and `__set__()` with the `__set__()` raising an `AttributeError` when called. Defining the `__set__()` method with an exception raising placeholder is enough to make it a data descriptor.

Note: descriptors are standard classes that just implement a specific set of methods, they can also contain anything else used on standard Python classes. This is especially useful when defining `__init__()` on a descriptor class, so that you can customize descriptors for individual attributes.

I.4.3 Invoking Descriptors

- Descriptors are assigned to class attributes, and the special methods are called automatically when the attribute is accessed, and the method used depends on what type of access is being performed.

- A descriptor can be called directly by its method name. For example, `d.__get__(obj)`.

- Alternatively, it is more common for a descriptor to be invoked automatically upon attribute access. For example, `obj.d` looks up `d` in the dictionary of `obj`. If `d` defines the method `__get__()`, then `d.__get__(obj)` is invoked according to the precedence rules listed below.

- The details of invocation depend on whether `obj` is an object or a class.

- For objects, the machinery is in `object.__getattribute__()` which transforms `b.x` into `type(b).__dict__['x'].__get__(b, type(b))`. The implementation works through a precedence chain that gives data descriptors priority over instance variables, instance variables priority over non-data descriptors, and assigns lowest priority to `__getattr__()` if provided.

- For classes, the machinery is in `type.__getattribute__()` which transforms `B.x` into `B.__dict__['x'].__get__(None, B)`.

- The important points to remember are:

  - descriptors are invoked by the `__getattribute__()` method
  - overriding `__getattribute__()` prevents automatic descriptor calls
  - `__getattribute__()` is only available with new style classes and objects
  - `object.__getattribute__()` and `type.__getattribute__()` make different calls to `__get__()`.
  - data descriptors always override instance dictionaries.
  - non-data descriptors may be overridden by instance dictionaries.

- The object returned by `super()` also has a custom `__getattribute__()` method for invoking descriptors. The call `super(B,obj).m()` searches `obj.__class__.__mro__` for the base class A immediately following B and then returns `A.__dict__['m'].__get__(obj, A)`. If it is not a descriptor, `m` is returned unchanged. If not in the dictionary, `m` reverts to a search using `object.__getattribute__()`.

- The details above show that the mechanism for descriptors is embedded in the `__getattribute__()` methods for object, type, and super. Classes inherit this machinery when they derive from object or if they have a meta-class providing similar functionality. Likewise, classes can turn-off descriptor invocation by overriding `__getattribute__()`.

I.4.5 Descriptor Example

- The following code creates a class whose objects are data descriptors which print a message for each `get()` or `set()`.

- Overriding `__getattribute__()` is an alternate approach that could do this for every attribute. However, this descriptor is useful for monitoring just a few chosen attributes:

- The simplest way to store a value for a descriptor takes advantage of a subtle distinction of how Python accesses values on an instance object. Every Python object has a namespace that's separate from the namespace of its class, so that each object can have different values attached to it. Normally, the object's attributes are a direct pass-through to this namespace, but descriptors short-circuit that process. Thankfully, Python allows another way to access the object's namespace directly: the `__dict__` attribute of the object.

- Every object has a `__dict__` attribute, which is a standard Python dictionary containing mappings for the various values attached to it. Even though descriptors get in the way of how this is normally accessed, your code can use `__dict__` to get at it directly, and it's a great place to store a single value.

- In the descriptor, we have to assign the value to the dictionary using a name, and the only way we know what name to use is to supply it explicitly. For this example, the constructor takes a required name argument, which will be used for the dictionary's key.

Note: if the value being retrieved isn't set, the expected behavior is to raise an `AttributeError`.

```
class SimpleDescriptor(object):
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        print("Call to get: {}".format(self.name))
        if self.name not in instance.__dict__:
            raise AttributeError(self.name)
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        print("Call  to  set  value:  {}->{}".format(self.name, 
value))
        instance.__dict__[self.name] = value
```

```
class Point:
    x=SimpleDescriptor("x")
    y=SimpleDescriptor("y")
    def __init__(self, x, y):
        self.x=x
        self.y=y
    def __str__(self):
        return "<{},{}>".format(self.x,self.y)

p1=Point(10,20)
p1.x = p1.x + 2
print("p1.y is:", p1.y)

p2=Point(1,1)
p2.y=100

print("p2.y is:", p2.y)
print("p1 is:", p1)
print("p2 is:", p2)
```

Output:

```
Call to set value: x->10
Call to set value: y->20
Call to get: x
Call to set value: x->12
Call to get: y
p1.y is: 20
Call to set value: x->1
Call to set value: y->1
Call to set value: y->100
Call to get: y
p2.y is: 100
p1 is: Call to get: x
Call to get: y
<12,20>
p2 is: Call to get: x
Call to get: y
<1,100>
```

I.5 Decorators

I.5.1 Overview

- Decorators are Python objects that can register, annotate, and/or wrap a Python function, method or class. Decorators appear in Python version 2.4.

- For example, the Python `atexit` module contains a `register()` function that registers a callback to be invoked when a Python program is exited.

- Without the decorator feature, a program that uses this function looks something like:

```
import atexit

def goodbye():
    print("Goodbye, world!")

atexit.register(goodbye)
```

- The decorator version looks like this:

```
import atexit

@atexit.register
def goodbye():
    print("Goodbye, world!")
```

- In the same way, to declare a class method, you can write:

```
class Test:
    def display(cls):
        print("Hello")
    display=classmethod(display)
```

- Or using a decorator:

```
class Test:
    @classmethod
    def display(cls):
        print("Hello")
```

- What are the advantages of the decorator syntax?

    1. You are made aware that the function is class method before you read the method body, giving you a better context for understanding the method.
    2. The method name is not repeated.

I.5.2 Creating Decorators

- Decorators may appear before any function definition, inner function or method definition.
- You can stack multiple decorators on the same function definition, one per line.
- A decorator is a callable object (like a function) that accepts one argument the function being decorated.
- The return value of the decorator replaces the original function definition.
- When you write:

```
@dec2
@dec1
def func(arg1, arg2, ...):
    pass
```

- This is equivalent to:

```
def func(arg1, arg2, ...):
    pass
func = dec2(dec1(func))
```

- Decorator are often used to annotate the function (by adding attributes to it), or wrapping the function with another function, then returning the wrapper (which replaces the original function).

Note: The decorator may return any kind of objects, which means that for advanced uses, you can turn functions or methods into specialized objects of your own choosing.

I.5.3 Classes as decorator

- A decorator doesn't have to be a function; it can be a class, as long as it can be called with a single argument (this will return a new instance of that class). Such a decorator will transform a function into an instance of a class.

- The following `traced` class can be used as a decorator:

```
class traced:
    def __init__(self,func):
        self.func = func

    def __call__(self,*__args,**__kw):
        print("entering", self.func)
        try:
            return self.func(*__args,**__kw)
        finally:
            print("exiting", self.func)
```

- After having decorated the following `hello()` function with `@traced` `hello` is no longer a function, but is instead an instance of the `traced` class that has the old `hello` function saved in its `func` attribute.

```
@traced
def hello():
    print("Hello, world!")

hello()
```

- When that wrapper instance is called using the statement `hello()` (a `traced` instance is a callable), Python's invokes the instance's `__call__()` method, which then invokes the original function between printing trace messages.

I.5.4 Stacking Decorators

- You can stack a decorator with another decorator.

- The ordering of the decorators determines the structure of the result.

- When using multiple decorators, you must know what kind of object each decorator expects to receive, and what kind of object it returns, so that the output of the innermost decorator is compatible with the input of the next-outer decorator.

- Usually, most decorators expect a function on input, and return either a function or an attribute descriptor as their output.

I.5.5 Functions as Decorators

- Most decorators expect an actual function as their input. To make our previous decorator (*traced*) which return a class instance be compatible with a wider range of decorators, we can modify it to return a function.

- The following decorator provides the same functionality as the original *traced* decorator, but instead of returning a traced object instance, it returns a new function object that wraps the original function.

```
def traced(func):
    def wrapper(*__args,**__kw):
        print("entering", func)
        try:
            return func(*__args,**__kw)
        finally:
            print("exiting", func)
    return wrapper
```

I.5.6 Inner functions and closures

- When you define a function inside of another function, any undefined local variables in the inner function will take the value of that variable in the outer function.

- In the preceding decorator, the value of *func* in the inner function comes from the value of *func* in the outer function.

- Because the inner function definition is executed each time the outer function is called, Python actually creates a new wrapper function object each time. Such function objects are called "***lexical closures***," because they enclose a set of variables from the lexical scope where the function was defined.

- A closure does not actually duplicate the code of the function, however. It simply encloses a reference to the existing code, and a reference to the free variables from the enclosing function. In this case, that means that the wrapper closure is essentially a pointer to the Python bytecode making up the wrapper function body, and a pointer to the local variables of the traced function during the invocation when the closure was created.

- Because a closure is really just a normal Python function object (with some predefined variables), and because most decorators expect to receive a function object, creating a closure is perhaps the most popular way of creating a stackable decorator.

I.5.7 Decorators with Arguments

- It is possible to pass one or more arguments to a decorator.
- When you write:

```
@decomaker(argA, argB, ...)
def func(arg1, arg2, ...):
    pass
```

- This is equivalent to:

```
func = decomaker(argA, argB, ...)(func)
```

- For instance you may decide to create a `@require` decorator that will give the possibility to record a method's precondition.
- Here is a simplistic version (inheritance is not taken into account for instance) of what this decorator may look like:

```
def require(expr):
    def decorator(func):
        def wrapper(*__args,**__kw):
            assert eval(expr),"Precond {} failed".format(expr)
            return func(*__args,**__kw)
        return wrapper
    return decorator

@require("len(__args)==1")
def test(*args):
    print(args[0])

test("Hello world!")
test(12,23) # AssertionError: Precond len(__args)==1 failed
```

- The `require` decorator creates two closures. The first closure creates a decorator function that knows the *`expr`* that was supplied to *`@require`*. This means `require` itself is not really the decorator function here. Instead, require returns the decorator function, here called `decorator`. This is very different from the previous decorators, and this change is necessary to implement parameterized decorators.
- The second closure is the actual wrapper function that evaluates *`expr`* whenever the original function is called.

Note: decorator invocations follow the same syntax rules as normal Python function or method calls, so you can use positional arguments, keyword arguments, or both.

I.5.8 Function Attributes

- Function attributes let you record arbitrary values as attributes on a function object.
  For example, suppose you want to track the author of a function or method, using an `@author` decorator. You could implement it this way:

```
def author(author_name):
    def decorator(func):
        func.author_name = author_name
        return func
    return decorator

@author("John Martin")
def sequenceOf(param1, param2):
    pass

print(sequenceOf.author_name)     # prints " John Martin "
```

- In this example, you simply set an `author_name` attribute on the function and return it, rather than creating a wrapper. Then, you can retrieve the attribute at a later time as part of some metadata-gathering operation.

I.5.9 Safe decorators

- It's easy to create a decorator that will work by itself, but creating a decorator that will work properly when combined with other decorators is a bit more complex.
  To the extent possible, a decorator should return an actual function object, with the same name and attributes as the original function, so as not to confuse an outer decorator or cancel out the work of an inner decorator.

- This means that decorators that simply modify and return the function they were given (like `@author`), are already safe. But decorators that return a wrapper function need to do two more things to be safe:
  1. Set the new function's name to match the old function's name.
  2. Copy the old function's attributes to the new function.

- The `@require` decorator has been modified to follow these recommendations:

```
def require(expr):
    def decorator(func):
        def wrapper(*__args,**__kw):
            assert eval(expr),"Precond {} failed".format(expr)
            return func(*__args,**__kw)
        wrapper.__name__ = func.__name__
        wrapper.__dict__ = func.__dict__
        wrapper.__doc__ = func.__doc__
        return wrapper
    return decorator
```

- The `functools` **`wraps()`** function is a facility that does exactly the same.

- `wraps()` takes a function used in a decorator and adds the functionality of copying over the function name, docstring, arguments list, etc.

- And since `wraps()` is itself a decorator, the following code does the correct thing:

```
def require(expr):
    def decorator(func):
        @wraps(func)
        def wrapper(*__args,**__kw):
            assert eval(expr),"Precond {} failed".format(expr)
            return func(*__args,**__kw)
        return wrapper
    return decorator
```

II TESTING PYTHON PROGRAMS

- Python developers have the choice between several unit test frameworks: unittest, doctest, py.test, ...

II.1 unittest

II.1.1 An overview

- The **`unittest`** module (referred to as **`PyUnit`** from time to time) has been part of the Python standard library since version 2.1.
- The `unittest` module is a Python language version of **`JUnit`**, a very popular Java testing framework.
- The framework implemented by `unittest` supports *fixtures*, *test suites*, and a *test runner* to enable automated testing for your code.
- Tests, as defined by `unittest`, have two parts: code to manage test “fixtures”, and the test itself.
- The canonical way of writing `unittest` tests is to derive a test class from **`unittest.TestCase`**.
- The test class exists in its own module, separate from the module containing the software to test.
- Test method names that start with "`test`" are automatically invoked by the framework.
- Each test method is executed independently from all other methods.
- Tests have 3 possible outcomes:

  `ok`

  The test passes.

  `FAIL`

  The test does not pass, and raises an `AssertionError` exception.

  `ERROR`

  The test raises an exception other than `AssertionError`.

- `TestCase` provides a **`setUp()`** method for setting up the fixture, and a **`tearDown()`** method for doing necessary clean-up.
- `TestCase` also provides custom assertions (for example **`assertEqual()`**, **`assertTrue()`** **`assertNotEqual()`**, …) that generate more meaningful error messages than the default Python assertions.

- Here is a short example of a test class for a module (stack.py) that define a Stack class.

```
import unittest
import stack

class TestStack(unittest.TestCase):
    """
    A test class for the stack module.
    """
    def setUp(self):
        """
        set up data used in the tests.
        setUp is called before each test function execution.
        """
        self.stack = stack.Stack(10)
        print("setUp is called")
    def tearDown(self):
        """
        tearDown is called after each test function execution.
        """
        print("tearDown is called")
    def testPush(self):
        self.stack.push(10)
        self.assertEqual(len(self.stack), 1)
        self.assertFalse(self.stack.isEmpty())
    def testPop(self):
        if len(self.stack) < self.stack.maxSize():
            val=23
            orgSize=len(self.stack)
            self.stack.push(23)
            self.assertEqual(self.stack.pop(), val)
            self.assertEqual(len(self.stack), orgSize)
    def testStackEmptyException(self):
        self.assertRaises(stack.StackEmptyError,
self.stack.pop)

if __name__ == '__main__':
     unittest.main()
```

```
setUp is called
tearDown is called
setUp is called
tearDown is called
setUp is called
tearDown is called
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

II.1.2 Test execution customization

- The canonical way of running tests in `unittest` is to include this code at the end of the module containing the test class:

```
if __name__ == '__main__':
    unittest.main()
```

- By default, `unittest.main()` builds a **`TestSuite`** object containing all the tests whose method names start with "test", then it invokes a **`TextTestRunner`** which executes each test method and prints the results to *stderr*.

Note: the order in which the tests are run is based on the alphanumerical order of their names, which can be sometimes annoying.

- Individual test cases can be run by simply specifying their names (prefixed by the test class name) on the command line:

```
# python test_stack.py testStack.testPush testStack.testPop
```

II.1.3 Test fixture management

- `TestCase` provides the **`setUp()`** and **`tearDown()`** methods that can be used in derived test classes in order to create/destroy "test fixtures", i.e. environments were data is set up so that each test method can act on it in isolation from all other test methods.

- In general, the `setUp/tearDown` methods are used for creating/destroying database connections, opening/closing files and other operations that need to maintain state during the test run.

- In my *`TestStack`* example, I'm using the `setUp()` method for creating a `Stack` object that can then be referenced by all test methods in my test class. Note that `setUp()` and `tearDown()` are called by the `unittest` framework before and after each test method is called. This ensures test independence, so that data created by a test does not interfere with data used by another test.

II.1.4 Test organization and reuse

- The `unittest` framework makes it easy to aggregate individual tests into test suites. There are several ways to create test suites. The easiest way is similar to this:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(unittest.makeSuite(TestStack))
    return suite
```

- Here a `TestSuite` object is create, then the `makeSuite()` helper function is used to build a test suite out of all tests whose names start with "test". The resulting suite is added to the initial `TestSuite` object via the `addTest()` method.

- A suite can also be created from individual tests by calling `addTest()` with the name of the test method (which in this case does not have to start with test):

```
suiteFew = unittest.TestSuite()
suiteFew.addTest(TestStack("testPush"))
suiteFew.addTest(TestStack("testPop"))
```

- In order to run a given suite, you can use a **`TextTestRunner`** object:

```
unittest.TextTestRunner().run(suiteFew)
# or
unittest.TextTestRunner(verbosity=2).run(suite())
```

- The first line runs a `TextTestRunner` with the default terse output and using the *`suiteFew`* suite, which contains only 2 tests.
  The second line increases the verbosity of the output, then runs the suite returned by the `suite()` method, which contains all tests starting with "test".

- The suite mechanism also allows for test reuse across modules: if you have defined 2 test modules with 2 distinct test suite you can aggregate the test suites:

```
import unittest
import TestStack
import TestVector

suite1 = TestVector.suite()
suite2 = TestStack.suite()

suite = unittest.TestSuite()

suite.addTest(suite1)
suite.addTest(suite2)
unittest.TextTestRunner(verbosity=2).run(suite)
```

II.1.5 Assertion syntax

- As previously mentioned, `unittest` provides its own custom assertions.
- Here are some of the reasons for this choice:
  - if tests are run with the optimization option turned on, the standard Python assert statements will be skipped; the unittest assert statements will not
  - the output of the standard Python assert statements does not show the expected and actual values that are compared the unittest assert statements do provide this information

II.1.5.1 Dealing with exceptions

- The TestStack class listed above has an example of testing for exceptions. In the `testStackEmptyException()` method, we test that calling pop on an empty `Stack` results in a `StackEmptyError` exception.
  The test will pass only when `StackEmptyError` is raised and will fail otherwise.
- You can implement the test function this way:

```
try:
     self.stack.pop()
except stack.StackEmptyError:
    pass
else:
    fail("Expected a StackEmptyError")
```

- A more concise way of testing for exceptions is to use the **`assertRaises()`** statement, passing it the expected exception type and the function/method to be called, followed by its arguments:

```
self.assertRaises(stack.StackEmptyError, self.stack.pop)
```

- All the assert methods (except **`assertRaises()`** and **`assertRaisesRegexp()`**) accept a *`msg`* argument that, if specified, is used as the error message on failure

II.1.5.2 Other assertion methods

- The `TestCase` class provides a number of methods to check for and report failures, such as:

Method	Checks that	New in
`assertEqual(a, b)`	a == b	
`assertNotEqual(a, b)`	a != b	
`assertTrue(x)`	bool(x) is True	
`assertFalse(x)`	bool(x) is False	
`assertIs(a, b)`	a is b	2.7
`assertIsNot(a, b)`	a is not b	2.7
`assertIsNone(x)`	x is None	2.7
`assertIsNotNone(x)`	x is not None	2.7
`assertIn(a, b)`	a in b	2.7
`assertNotIn(a, b)`	a not in b	2.7
`assertIsInstance(a, b)`	isinstance(a, b)	2.7
`assertNotIsInstance(a, b)`	not isinstance(a, b)	2.7

- There are also other methods used to perform more specific checks, such as:

Method	Checks that	New in
`assertAlmostEqual(a, b)`	round(a-b, 7) == 0	
`assertNotAlmostEqual(a, b)`	round(a-b, 7) != 0	
`assertGreater(a, b)`	a > b	2.7
`assertGreaterEqual(a, b)`	a >= b	2.7
`assertLess(a, b)`	a < b	2.7
`assertLessEqual(a, b)`	a <= b	2.7
`assertRegexpMatches(s, re)`	regex.search(s)	2.7
`assertNotRegexpMatches(s, re)`	not regex.search(s)	2.7
`assertItemsEqual(a, b)`	sorted(a) == sorted(b) and works with unhashable objs	2.7
`assertDictContainsSubset(a, b)`		

- Finally the `TestCase` provides the following methods and attributes:

  `fail(msg=None)`

  Signals a test failure unconditionally, with msg or None for the error message.

  `ongMessage`

  If set to True then any explicit failure message you pass in to the assert methods will be appended to the end of the normal failure message. The normal messages contain useful information about the objects involved, for example the message from `assertEqual()` shows you the *repr* of the two unequal objects. Setting this attribute to `True` allows you to have a custom error message in addition to the normal one.

II.1.6 Summary

- To summarize, here are some Pros and Cons of using the `unittest` framework.

  unittest Pros

  available in the Python standard library
  easy to use by people familiar with the JUnit frameworks
  flexibility in test execution via command-line arguments
  support for test fixture/state management via set-up/tear-down hooks
  strong support for test organization and reuse via test suites

  unittest Cons

  JUnit flavor may be too strong for "pure" Pythonistas
  tests are executed in alphanumerical order
  assertions use custom syntax

II.2 Unit test with the doctest module

- **`doctest`** lets you test your code by running examples embedded in the "docstrings" and verifying that they produce the expected results.

- `doctest` works by parsing the docstrings to find examples, running them, then comparing the output text against the expected value.

II.2.1 Getting Started

- The first step to setting up *doctests* is to use the interactive interpreter to create examples and then copy and paste them into the docstrings in your module.

- *doctest* lets you keep the surrounding text you would normally include in the docstrings. It looks for lines beginning with the interpreter prompt, >>>, to find the beginning of a test case. The case is ended by a blank line, or by the next interpreter prompt.

```
def my_function(a, b):
    """
    Returns a * b.
    Works with numbers:

    >>> my_function(2, 3)
    6

    and strings:

    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
```

- To run the tests, use `doctest` as the main program via the `-m` option of the interpreter. Usually no output is produced while the tests are running, so the example below includes the **`-v`** option to make the output more verbose.

```
$ python -m doctest -v module_to_test.py

Trying:
    my_function(2, 3)
Expecting:
    6
ok
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    module_to_test
1 items passed all tests:
   2 tests in module_to_test.my_function
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

II.2.2 Handling Unpredictable Output

- There are situations where the exact output of a function may not be predictable: for instance, a function may return a different date each time it is being called. *doctest* offer techniques for dealing with them.
- When the tests include values that are likely to change in unpredictable ways, and where the actual value is not important to the test results, you can use the **`ELLIPSIS`** option to tell *doctest* to ignore portions of the verification value.
- The comment after the call to `my_function()` (`#doctest: +ELLIPSIS`) tells `doctest` to turn on the `ELLIPSIS` option for that test. The ... replaces the random numeric value, so that portion of the expected value is ignored and the actual output matches and the test passes.

```
from random import randint
def my_function(a, b):
    """
    This is a function that makes use of the * operator
    >>> my_function(2, 3) #doctest: +ELLIPSIS
    (6, ...)
    >>> my_function('a', 3) #doctest: +ELLIPSIS
    ('aaa', ...)

    End of the docstring
    """
    return (a * b, randint(1,50))
```

II.2.3 Tracebacks

- Tracebacks are a special case of changing data. Since the paths in a `traceback` depend on the location where a module is installed on the filesystem on a given system, it would be impossible to write portable tests if they were treated the same as other output.
- `doctest` makes a special effort to recognize tracebacks: when doctest sees a traceback header line (either *`Traceback (most recent call last):`* or *`Traceback (innermost last):`*, depending on the version of Python you are running), it skips ahead to find the exception type and message, ignoring the intervening lines entirely.

```
def this_raises():
    """This function always raises an exception.

    >>> this_raises()
    Traceback (most recent call last):
    RuntimeError: here is the error
    """
    raise RuntimeError('here is the error')
```

II.2.4 Working Around Whitespace

- In real world applications, output usually includes whitespace such as blank lines, tabs, and extra spacing to make it more readable. Blank lines, in particular, cause issues with doctest because they are used to delimit tests.To match the blank lines, replace them in the sample input with the string **`<BLANKLINE>`**.

- Another pitfall of using text comparisons for tests is that embedded whitespace can also cause tricky problems with tests. The following example has a single extra space after the 6. Extra spaces can find their way into your code via copy-and-paste errors, but since they come at the end of the line, they can go unnoticed in the source file and be invisible in the test failure report as well.

- Using one of the diff-based reporting options, such as **`REPORT_NDIFF`**, shows the difference between the actual and expected values with more detail, and the extra space becomes visible.

```
def my_function(a, b):
    """
    >>> my_function(2, 3) #doctest: +REPORT_NDIFF
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
```

- Unified (`REPORT_UDIFF`) and context (`REPORT_CDIFF`) diffs are also available, for output where those formats are more readable.

```
$ python -m doctest -v doctest_ndiff.py

Trying:
    my_function(2, 3) #doctest: +REPORT_NDIFF
Expecting:
    6
**************************************************************
********
File "doctest_ndiff.py", line 12, in doctest_ndiff.my_function
Failed example:
    my_function(2, 3) #doctest: +REPORT_NDIFF
Differences (ndiff with -expected +actual):
    - 6
    ?  -
    + 6
Trying:
    my_function('a', 3)
Expecting:
    'aaa'
Ok
```

```
1 items had no tests:
    doctest_ndiff
*************************************************************
********
1 items had failures:
   1 of   2 in doctest_ndiff.my_function
2 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

- There are cases where it is beneficial to add extra whitespace in the sample output for the test, and have doctest ignore it. For example, data structures can be easier to read when spread across several lines, even if their representation would fit on a single line.

- When **`NORMALIZE_WHITESPACE`** is turned on, any whitespace in the actual and expected values is considered a match. You cannot add whitespace to the expected value where none exists in the output, but the length of the whitespace sequence and actual whitespace characters do not need to match. The first test example gets this rule correct, and passes, even though there are extra spaces and newlines. The second has extra whitespace after [and before], so it fails.

```
def my_function(a, b):
    """Returns a * b.

    >>>    my_function(['A',    'B',    'C'],    3)    #doctest:
+NORMALIZE_WHITESPACE
    ['A', 'B', 'C',
     'A', 'B', 'C',
     'A', 'B', 'C']

    This does not match because of the extra space after the [
in the list

    >>>    my_function(['A',    'B',    'C'],    2)    #doctest:
+NORMALIZE_WHITESPACE
    [ 'A', 'B', 'C',
      'A', 'B', 'C' ]
    """
    return a * b
```

II.2.5 Test Locations

- All of the tests in the examples so far have been written in the docstrings of the functions they are testing. But with doctest tests every docstring can contain tests at the module, class and function level.

- In cases you want to put your tests elsewhere than within docstrings. doctest also looks for a module-level variable called **`__test__`** and uses it to locate other tests. `__test__` should be a dictionary mapping test set names (as strings) to strings, modules, classes, or functions.

```
import doctest_private_tests_external

__test__ = {
    'numbers':"""
>>> my_function(2, 3)
6

>>> my_function(2.0, 3)
6.0
""",

    'strings':"""
>>> my_function('a', 3)
'aaa'

>>> my_function(3, 'a')
'aaa'
""",

    'external':doctest_private_tests_external,

    }

def my_function(a, b):
    """Returns a * b
    """
    return a * b
```

- If the value associated with a key is a string, it is treated as a docstring and scanned for tests. If the value is a class or function, doctest searchs them recursively for docstrings, which are then scanned for tests.

II.2.6 External Documentation

- Mixing tests in with your code isn't the only way to use *doctest*. Examples embedded in external project documentation files, such as `reStructuredText` files, can be used as well.

```
# This is the file "myTest.py"
def my_function(a, b):
    """Returns a*b
    """
    return a * b
```

The help for `myTest` is saved to a separate file, `myTest.rst` (see below). The examples illustrating how to use the module are included with the help text, and *doctest* can be used to find and run them.

```
===============================
 How to Use MyTest.py
===============================

This library is very simple, since it only has one function
called
``my_function()``.

Numbers
=======

``my_function()`` returns the product of its arguments.  For
numbers, that value is equivalent to using the ``*`` operator.

::

    >>> from doctest_in_help import my_function
    >>> my_function(2, 3)
    6

It also works with floating point values.

::

    >>> my_function(2.0, 3)
    6.0

Non-Numbers
===========

Because ``*`` is also defined on data types other than
numbers,``my_function()`` works just as well if one of the
arguments is a string, list, or tuple.
```

```
::

    >>> my_function('a', 3)
    'aaa'

    >>> my_function(['A', 'B', 'C'], 2)
    ['A', 'B', 'C', 'A', 'B', 'C']
```

- The tests in the text file can be run from the command line, just as with the Python source modules.

```
$ python -m doctest -v myTest.rst
```

- Normally doctest sets up the test execution environment to include the members of the module being tested, so your tests don't need to import the module explicitly. In this case, however, the tests aren't defined in a Python module, doctest does not know how to set up the global namespace, so the examples need to do the import work themselves. All of the tests in a given file share the same execution context, so importing the module once at the top of the file is enough.

II.2.7 Running Tests

- The previous examples all use the command line test runner built into `doctest`. It is easy and convenient for a single module, but will quickly become tedious as your package spreads out into multiple files. There are several alternative approaches.

II.2.7.1 By Module

- You can include instructions to run doctest against your source at the bottom of your modules. Use **`testmod()`** without any arguments to test the current module.

```
def my_function(a, b):
    """
    >>> my_function(2, 3)
    6
    >>> my_function('a', 3)
    'aaa'
    """
    return a * b

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

- The first argument to `testmod()` is a module containing code to be scanned for tests. This feature lets you create a separate test script that imports your real code and runs the tests in each module one after another.

```
import myTest

if __name__ == '__main__':
    import doctest
    doctest.testmod(myTest)
```

- You can build a test suite for your project by importing each module and running its tests.

II.2.7.2 By File

- **`testfile()`** works in a way similar to `testmod()`, allowing you to explicitly invoke the tests in an external file from within your test program.

```
import doctest

if __name__ == '__main__':
    doctest.testfile('myTest.rst')
```

II.2.7.3 Unittest Suite

- If you use both `unittest` and `doctest` for testing the same code in different situations, you may find the unittest integration in doctest useful for running the tests together. Two classes, **`DocTestSuite`** and **`DocFileSuite`** create test suites compatible with the test-runner API of `unittest`.

```
import doctest
import unittest

import doctest_simple

suite = unittest.TestSuite()
suite.addTest(doctest.DocTestSuite(doctest_simple))
suite.addTest(doctest.DocFileSuite('doctest_in_help.rst'))

runner = unittest.TextTestRunner(verbosity=2)
runner.run(suite)
```

- The tests from each source are collapsed into a single outcome, instead of being reported individually.

II.2.8 Test Context

- The execution context created by `doctest` as it runs tests contains a copy of the module-level globals for the module containing your code. This isolates the tests from each other somewhat, so they are less likely to interfere with one another. Each test source (function, class, module) has its own set of global values.

```
class TestGlobals(object):

    def one(self):
        """
        >>> var = 'value'
        >>> 'var' in globals()
        True
        """

    def two(self):
        """
        >>> 'var' in globals()
        False
        """
```

- `TestGlobals` has two methods, `one()` and `two()`. The tests in the docstring for `one()` set a global variable, and the test for `two()` looks for it (expecting not to find it).

- That does not mean the tests cannot interfere with each other, though, if they change the contents of mutable variables defined in the module.

- If you need to set global values for the tests, to parameterize them for an environment for example, you can pass values to `testmod()` and `testfile()` and have the context set up using data you control.

II.2.9 doctest vs unittest

- Many developers find `doctest` easier to use than `unittest` because, in its simplest form, there is no API to learn before using it. However, as the examples become more complex the lack of fixture management can make writing `doctest` tests more cumbersome than using `unittest`.

III REGULAR EXPRESSIONS

III.1 Overview

- The **`re`** module provides Perl-style regular expression patterns.
- Regular expressions (RE) are being used to test if a given string match a pattern, or to test if there a match for the pattern anywhere in this string. You can also use REs to modify a string or to split it apart in various ways.
- Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C.

Note: the regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions.

III.2 Regular expressions syntax

- For a string to match a regular expression, they must be identical, with the exception of special metacharacters that may occur in the pattern specifier.
- Here's a complete list of the metacharacters; their meanings will be discussed in the rest of this chapter.

```
. ^ $ * + ? { [ ] \ | ( )
```

- You can escape a metacharacter by preceding it with a backslash.

Note: to match a literal backslash, one has to write '\\\\' as the RE string, because the regular expression must be "\\", and each backslash must be expressed as "\\" inside a regular Python string literal. To avoid too many \, a solution is to use Python's raw string notation for regular expressions: `r`"\\"

III.2.1 The main metacharacters

- The following table describes some of these metacharacters:

`.`	Matches any character.
`*`	Indicates zero or more matches of the previous item.
`+`	Indicates one or more matches of the previous item.
`|`	Indicates a choice between two expressions.
`[...]`	Matches any character in a set.
`^`	Matches the start of the string.
`$`	Matches the end of the string.
`\<char>`	Matches the otherwise special character <char>; e.g., * matches a *.

- The metacharacter `.`, matches anything except a newline character.
  There is an alternate mode (**`re.DOTALL`**) where it will match even a newline.

- Usually **^** matches only at the beginning of the string, and **$** matches only at the end of the string and immediately before the newline (if any) at the end of the string.

- When the flag (**re.MULTILINE**) is specified, ^ matches at the beginning of the string and at the beginning of each line within the string, immediately following each newline. Similarly, the $ metacharacter matches either at the end of the string and at the end of each line (immediately preceding each newline).

- A vertical bar **|** represents an "or" and parentheses **(...)** can be used to group things together:

```
jelly|cream  # Either jelly or cream
(eg|le)gs    # Either eggs or legs
(si)+        # Either si or sisi or sisisi or...
```

III.2.2 Character classes

- Square brackets **[...]** are used to match any one of the characters inside them. Inside square brackets a **-** indicates "between" and a **^** at the beginning means "not":

[qjk]	Either q or j or k
[^qjk]	Neither q nor j nor k
[a-z]	Anything from a to z inclusive
[^a-z]	No lower case letters
[a-zA-Z]	Any letter
[a-z]+	Any non-zero sequence of lower case letters

- Metacharacters are not active inside square brackets. For example, [abc$] will match any of the characters "a", "b", "c", or "$".

- In addition:

```
\w      # matches an alphanumeric character (including "_")
\W      # matches a non-alphanumeric character
\s      # matches a whitespace character
\S      # matches a non-whitespace character
\d      # matches a numeric character
\D      # matches a non-numeric character
\A      # matches at the start of a string
\Z      # matches at the end of a string
\b      # matches the empty string at the beginning or end
        # of a word
\B      # matches the empty string if it is not at
        # the beginning or at the end of a word
```

Note: the backslashed special characters (\n, \f, \t, ...) have their default meaning.

- You may use \w, \s and \d within character classes (square brackets).

- Within character classes `\n`, `\r`, `\f`, `\t`, `\a`, `\b` and `\xNNN` have their normal interpretations (newline, carriage return, formfeed, horizontal tab, alert, backspace and the character whose hexadecimal value is NNN respectively).

III.2.3 Repetition factors

- Any item of a regular expression may be followed with digits in curly brackets of the form **`{n,m}`**, where *`n`* gives the minimum number of times to match the item and *`m`* gives the maximum.
- *`n`* an *`m`* are unsigned decimal integers with permissible values from 0 to 255.
- The form **`{n}`** is equivalent to `{n,n}` and matches exactly *`n`* times.
- The form **`{n,}`** matches *`n`* or more times.
- If a curly bracket occurs in any other context, it is treated as a regular character.
- The `*` modifier is equivalent to *`{0,}`*, the `+` modifier to *`{1,}`* and the `?` modifier to *`{0,1}`*.

III.2.4 Greedy/non-greedy match

- In the event that an regular expression could match more than one substring of a given string, the RE matches the one starting earliest in the string.
  If the RE could match more than one substring starting at that point, its choice is determined by its ***preference***: either the longest substring, or the shortest.
- Repetitions such as * are *greedy*; when repeating a RE, the matching engine will try to repeat it as many times as possible. If later portions of the pattern don't match, the matching engine will then back up and try again with few repetitions.

```
*?
+?
??
{m}?
{m,}?
{m,n}?
```

match the same possibilities as `*`, `+`, `?`, `{m}`, `{m,}`, `{m,n}`, but prefer the smallest number rather than the largest number of matches.

III.2.5 Groups

- Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest.

- You can identify groups (subexpressions) in a regular expression by surrounding them in parentheses **`()`**.

- You can repeat the contents of a group with a repeating qualifier, such as `*`, `+`, `?`, or `{m,n}`. For example, `(ab)*` will match zero or more repetitions of "`ab`".

- You can also use groups for extracting subexpressions in a match.
  Groups can be retrieved by passing an argument to `group()`, `start()`, `end()`, and `span()`. Groups are numbered starting with 0. Group 0 is always present; it's the whole RE, so `MatchObject` methods all have group 0 as their default argument.

- Subgroups are numbered from left to right, from 1 upward. Groups can be nested; to determine the number, just count the opening parenthesis characters, going from left to right.

- Within a pattern, **`\`*`grp-nb`*** (where *`grp-nb`* is a nonzero digit) matches the same string matched by the parenthesized subexpression specified by the number *`grp-nb`*.

- Sometimes you'll want to use a group to collect a part of a regular expression, but aren't interested in retrieving the group's contents. You can make this fact explicit by using a **non-capturing group**: **`(?:...)`**, where you can put any other regular expression inside the parentheses.

- Instead of referring to groups by numbers, groups can be referenced by a name. The syntax for a **named group** is one of the Python-specific extensions:

  `(?P<`*`name`*`>...)`

- *`name`* is the name of the group. You can put any other regular expression inside the parentheses.

- Within a pattern, **`(?P=`*`name`*`)`** (where *name* is a group name) matches the same string matched by the group *`name`*

III.2.6 Lookahead Assertions

- Lookahead assertions are available in both positive and negative form, and look like this:

`(?=...)`

Positive lookahead assertion. This succeeds if the contained regular expression (represented here by ...) successfully matches at the current location, and fails otherwise. But, once the contained expression has been tried, the matching engine doesn't advance at all; the rest of the pattern is tried right where the assertion started.

`(?!...)`

Negative lookahead assertion. This is the opposite of the positive assertion; it succeeds if the contained expression doesn't match at the current position in the string.

- The following regular expression matches a time given in the format: hhHmm except if mm is 00.

```
regexp=re.compile(r"(\d\d)H(?!00$)(\d\d$)")
```

III.3 The re module

- The **`re`** module provides an interface to the regular expression engine, allowing you to compile REs into objects and then perform matches with them.

III.3.1 Compiling Regular Expressions

- Regular expressions are compiled into **`RegexObject`** instances, which have methods for various operations such as searching for pattern matches or performing string substitutions.

- To compile a regular expression, you use the **`compile(pattern [,flag])`** function.

```
import re
regexp=re.compile("^\d")
text=input("Enter a string: ")
if regexp.match(text):
    print("Match")
else:
    print("Does not match")
```

- The RE is passed to `compile()` as a string.

- The `compile()` method also accepts an optional flags argument, used to enable various special features and syntax variations.

- The compilation flags are available in the `re` module under two names, a long name, and a short, one-letter form.

- Multiple flags can be specified by bitwise OR-ing them.

- Here's a table of the available flags:

Flag	Shortcut	Meaning
`DOTALL`	**`S`**	Make . match any character, including newlines
`IGNORECASE`	**`I`**	Do case-insensitive matches
`LOCALE`	**`L`**	Make \w, \W, \b, and \B, dependent on the current locale.
`MULTILINE`	**`M`**	Multi-line matching, affecting ^ and $
`VERBOSE`	**`X`**	Enable verbose REs, which can be organized more cleanly and understandably.

- The VERBOSE flag allows you to write regular expressions that are more readable by granting you more flexibility in how you can format them.

- When this flag has been specified, whitespace within the RE string is ignored, except when the whitespace is in a character class or preceded by an unescaped backslash; this lets you organize and indent the RE more clearly.

- It also enables you to put comments within a RE that will be ignored by the engine; comments are marked by a "#" that is neither in a character class or preceded by an unescaped backslash.

- The following pattern matches any valid integer literal represented in decimal.

```
regexp=re.compile("^[+-]?([1-9]\d*|0)$")
```

using the VERBOSE flag you can write the pattern this way:

```
regexp=re.compile(r"""^        # beginning of the string
                   [+-]?       # an optional sign
                   ([1-9]\d*   # one or more digit
                    |0)        # 0
                    $"""       # end of the string
                    , re.VERBOSE)
```

III.3.2 Performing Matches

- RegexObject instances have several methods and attributes.

- The most significant ones are:

match()	Determine if the RE matches at the beginning of the string.
search()	Scan through a string, looking for any location where this RE matches.
findall()	Find all substrings where the RE matches, and returns them as a list.
finditer()	Find all substrings where the RE matches, and returns them as an iterator.

- match() and search() return None if no match can be found.
  If they're successful, a MatchObject instance is returned, containing information about the match: where it starts and ends, the substring it matched, and more.

- You can query the MatchObject for information about the matching string. MatchObject instances have several methods and attributes; the most important ones are:

group()	Return the string matched by the RE
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match

- The `MatchObject` methods that deal with capturing groups all accept either integers, to refer to groups by number, or a string containing the group name (if named group have been used in the regular expression).

```
import re

regexp=re.compile("[+-]?([1-9]\d*|0)")

nb=input("Enter a string: ")

mo= regexp.match(nb)
if (mo):
    print("Match")
    print(mo.group())
    print(mo.start(), mo.end())
    print(mo.span())
else:
    print("Does not match")
```

```
Enter a string: 346
Match
346
0 3
(0, 3)
```

Note: since the match method only checks if the RE matches at the start of a string, `start()` will always be zero. However, the `search()` method of `RegexObject` instances scans through the string, so the match may not start at zero in that case.

```
import re

regexp=re.compile("[+-]?([1-9]\d*|0)")

nb=input("Enter a string: ")

mo= regexp.search(nb)
if (mo):
    print("Match")
    print(mo.group())
    print(mo.start(), mo.end())
    print(mo.span())
else:
    print("Does not match")
```

```
Enter a string: 346
Match
346
6 9
(6, 9)
```

- Groups indicated with **`( )`** also capture the starting and ending index of the text that they match; this can be retrieved by passing an argument to `group()`, `start()`, `end()`, and `span()`.

- `group()` can be passed multiple group numbers at a time, in which case it will return a tuple containing the corresponding values for those groups.

- The **`groups()`** method returns a tuple containing the strings for all the subgroups, from 1 up to how many there are.

```
import re
regexp=re.compile(r"(\d\d?)H(\d\d?)")
val=input("Enter a string: ")
mo= regexp.search(val)
if (mo):
    print("Match")
    print(mo.groups())
    print(mo.group(0))
    print(mo.group(1))
    print(mo.group(2))
else:
    print("Does not match")
```

```
Enter a string: 23H45
Match
('23', '45')
23H45
23
45
```

- The **`findall()`** method returns a list of matching strings:

```
>>> p = re.compile('\d+')
>>> p.findall(' ... 10 ... 23 ... 34 ')
['10', '23', '34']
```

- The **`finditer()`** method returns a sequence of `MatchObject` instances as an ***iterator***.

```
>>> p = re.compile('\d+')
>>> p.findall(' .. 10 .. 23 .. 34 ')
['10', '23', '34']
>>> it= p.finditer(' .. 10 .. 23 .. 34 ')
>>> for m in it:
...     print(m.span())
...
(4, 6)
(10, 12)
(16, 18)
```

III.3.3 Module-Level Functions

- You don't have to produce a `RegexObject` using a `compile()` method and call its methods; the `re` module also provides top-level functions called **`match()`**, **`search()`**, **`sub()`**, and so forth.

- These functions take the same arguments as the corresponding `RegexObject` method, with the RE string added as the first argument, and still return either `None` or a `MatchObject` instance.

III.3.4 Modifying Strings

- Regular expressions are also commonly used to modify a string in various ways, using the following `RegexObject` methods:

`split()`	split the string into a list, splitting it wherever the RE matches
`sub()`	find all substrings where the RE matches, and replace them with a different string
`subn()`	does the same thing as `sub()`, but returns the new string and the number of replacements

III.3.5 Splitting Strings

- The **`split()`** method of a `RegexObject` splits a string apart wherever the RE matches, returning a list of the pieces.

  `split( string [, maxsplit = 0])`

- If capturing parentheses are used in the RE, then their contents will also be returned as part of the resulting list. Thanks to that, you can retrieve the text between delimiters but also the delimiter themselves.

- You can limit the number of splits made, by passing a value for `maxsplit`.

- This method is similar to the `split()` method of strings but provides much more generality in the delimiters that you can split by.

```
import re
regexp=re.compile(r"(\W+)")
result=regexp.split("nb += 23; var=56;")
print(result)
```

```
['nb', ' += ', '23', '; ', 'var', '=', '56', ';', '']
```

III.3.6 Search and Replace

- Another common task is to find all the matches for a pattern, and replace them with a different string. The **`sub()`** method takes a replacement value, which can be either a string or a function, and the string to be processed.

  `sub(replacement, string[, count = 0])`

- This method returns the string obtained by replacing the leftmost non-overlapping occurrences of the RE in *`string`* by the string *`replacement`*. If the pattern isn't found, *`string`* is returned unchanged.

- The optional argument *`count`* is the maximum number of pattern occurrences to be replaced; `count` must be a non-negative integer. The default value of 0 means to replace all occurrences.

```
import re

regexp=re.compile(r"\d+")

print(regexp.sub("NB", "var = 12; nb = 34"))
```

```
var = NB; nb = NB
```

- The **`subn()`** method does the same work, but returns a tuple containing the new string value and the number of replacements that were performed:

```
import re
regexp=re.compile(r"\d+")
print(regexp.subn("NB", "var = 12; nb = 34"))
```

```
('var = NB; nb = NB', 2)
```

- In the *`replacement`* string, reference to group number such as "**`\`*`grp-nb`***" or to a group name such as "**`\g<`*`name`*`>`**", are replaced with the substring matched by the corresponding group in the RE.

```
import re
regexp=re.compile(r"(\d+):(\d+)")
print(regexp.sub(r"\2:\1", "254:678"))
```

```
678:254
```

- If *`replacement`* is a function, the function is called for every non-overlapping occurrence of pattern. On each call, the function is passed a `MatchObject` argument for the match and can use this information to compute the desired replacement string and return it.

IV INTERACTION WITH THE OPERATING SYSTEM

IV.1 The sys module

- The **`sys`** module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

- Here are some interesting attributes and methods provided by this module.

`argv`

The list of command line arguments passed to a Python script.
`argv[0]` is the script name

`builtin_module_names`

A tuple of strings giving the names of all modules that are compiled into this Python interpreter.

`exit([arg])`

Exit from Python. This is implemented by raising the `SystemExit` exception. The optional argument arg can be an integer giving the exit status (defaulting to zero).

`getrefcount(object)`

Return the reference count of the object.

`maxint`

The largest positive integer supported by Python's regular integer type.

`modules`

This is a dictionary that maps module names to modules which have already been loaded.

`path`

A list of strings that specifies the search path for modules. Initialized from the environment variable **`PYTHONPATH`**, plus an installation-dependent default.

`ps1`
`ps2`

Strings specifying the primary and secondary prompt of the interpreter.

`stdin`
`stdout`
`stderr`

File objects corresponding to the interpreter's standard input, output and error streams.

`version`

A string containing the version number of the Python interpreter plus additional information on the build number and compiler used.

IV.2 The OS module

- The **`os`** module presents a (reasonably) uniform cross-platform view of the different capabilities of various operating systems.
- The module provides functionality for creating files, manipulating files and directories, and creating, managing, and destroying processes.

IV.2.1 OS name

- The `os` module supplies a **`name`** attribute, which is a string that identifies the kind of platform on which Python is being run (posix, nt, mac, …).

IV.2.2 OSError Exceptions

- When a request to the operating system fails, `os` raises an exception, an instance of **`OSError`**.
- Instances of `OSError` expose three useful attributes:

  `errno` : The numeric error code of the operating system error

  `strerror` : A string that summarily describes the error

  `filename` : The name of the file on which the operation failed (for file-related functions only)

IV.2.3 Handling of the FileSystem

- Using the **`os`** module, you can manipulate the filesystem in a variety of ways: creating, copying, and deleting files and directories, comparing files, and examining filesystem information about files and directories.
- Several other related modules operate on the filesystem: **`os.path`**, **`os.stat`**, **`filecmp`**, **`shutil`**

IV.2.3.1 File path (the os.path module)

- On both Unix-like and Windows platforms, Python accepts Unix syntax for paths, with slash (/) as the directory separator.
- On non-Unix-like platforms, Python also accepts platform-specific path syntax.
- Module `os` supplies attributes that provide details about path strings on the current platform (`curdir`, `defpath`, `linesep`, `extsep`, `pathsep`, `pardir`, `sep`), but higher-level path manipulation operations are provided by the module **`os.path`**.
- The **`os.path`** module supplies functions to analyze and transform path strings. The most important ones are presented below

`basename(path)`

Returns the base name part of path.

```
>>> from os.path import basename
>>> basename("/users/peter/cmd.txt")
'cmd.txt'
```

`dirname(path)`

Returns the directory part of path

```
>>> from os.path import dirname
>>> dirname("/users/peter/cmd.txt")
'/user/peter'
```

`exists(path)`

Returns True when path names an existing file or directory, otherwise False.

`expandvars(path)`

Returns a copy of string path, replacing each substring of the form "`$name`" or "`${name}`" with the value of environment variable *`name`*.

`getsize(path)`

Returns the size of the corresponding file.

`isfile(path)`
`isdir(path)`
`islink(path)`

Returns True when path names respectively: an existing regular file, an existing directory, or a symbolic link.

`join(path[, path, …])`

Returns a string that joins the argument strings with the appropriate path separator for the current platform.

```
>>> os.path.join('/users', 'peter', 'bin/shells')
/users/peter/bin/shells
```

`split(path)`

Returns a pair of strings (dir, base) such that join(dir, base) equals path.

```
>>> os.path.split('/users/peter/cmd.txt')
('/users/peter', 'cmd.txt')
```

IV.2.3.2 File and Directory Functions of the os Module

- The `os` module supplies several functions to query and set file and directory status.

`access(path,mode)`

Returns `True` if file *`path`* has all of the permissions encoded in integer *`mode`* (`os.F_OK`, `os.R_OK`, `os.W_OK` or `os.X_OK`) , otherwise `False`. The mode can be combined using the bitwise-OR operator |).

`chdir(path)`

Sets the current working directory to path.

`chmod(path,mode)`

Changes the permissions of file *`path`*, as encoded in integer *`mode`*.

`getcwd()`

Returns the path of the current working directory.

`listdir(path)`

Returns a list whose items are the names of all files and subdirectories found in directory path (the special directory names '.' and '..' are not included).

`makedirs(path,mode=0777)`
`mkdir(path,mode=0777)`

`makedirs` creates all directories that are part of path and do not yet exist. `mkdir` creates only the rightmost directory of path. Both functions use `mode` as permission bits of directories they create.

`remove(path)`
`unlink(path)`

Removes the file named path.

`rename(source,dest)`

Renames the file or directory named source to dest.

`rmdir(path)`

Removes the empty directory named path.

`stat(path)`

Like the C `stat()` function, returns a tuple of 10 integers that provide information about a file or subdirectory path (Protection mode, Inode nb, Nb of link, Device ID, UID, GID, Size, Time of last access, Time of last modification, Time of last status change).

`tempnam(dir=None,prefix=None)`

Returns an absolute path usable as the name of a new temporary file.

`walk(root[, topdown=True[, onerror=None[, followlinks=False]]])`

`root`: the root directory
`topdown`: specified if directories are scanned from top-down (True, the default) or bottom-up.
`onerror`: by default error are ignored, but you can provide a function to be exceuted when error are encountered.
`followlinks`: visits directories pointed to by symlinks (if True).

`walk()` returns a tuple composed of three items: the root directory, a list of directories (dirs) immediately below the current root and a list of files found in those directories. In the next iteration, each directory of those in the previous dirs list will take on the role of root in turn and the search will continue from there, going down a level only after the current level has been searched.

A typical use of `walk()` is to print all files and subdirectories in a tree:

```
import os

# traverse root directory, and list directories as dirs and
files as files
for root, dirs, files in os.walk("c:/pierrefo"):
    path = root.split('/')
    print((len(path) - 1) * '---', os.path.basename(root))
    for file in files:
        print(len(path) * '---', file)
```

IV.2.3.3 The shutil Module

- The **`shutil`** module (an abbreviation for shell utilities) supplies functions to copy files and to remove an entire directory tree.

`copy(src,dst)`

Copies the contents of file *`src`*, creating or overwriting file *`dst`*. If *`dst`* is a directory, the target is a file with the same base name as *`src`* in directory *`dst`*.

`copyfile(src,dst)`

Copies the contents only of file *`src`*, creating or overwriting file *`dst`*.

`copyfileobj(fsrc,fdst,bufsize=16384)`

Copies `file` object *`fsrc`*, which must be open for reading, to `file` object *`fdst`*, which must be open for writing. Copies no more than *`bufsize`* bytes at a time if *`bufsize`* is greater than 0.

`copytree(src,dst,symlinks=False)`

Copies the whole directory tree rooted at *`src`* into the destination directory named by *`dst`*. *`dst`* must not already exist, as copytree creates it. When *`symlinks`* is true, copytree creates symbolic links in the new tree when it finds symbolic links in the source tree.

`rmtree(path)`

Removes the directory tree rooted at path.

IV.2.4 Running Other Programs

- The **`os`** module offers several ways for your program to run other programs.

- The simplest way to run another program is through function **`os.system()`**, although this offers no way to control the external program.

- The `os` module also provides a number of **`exec`** functions which offer fine-grained control.

- A program run by one of the `exec` functions, however, replaces the current program (i.e., the Python interpreter) in the same process.

- Finally, `os` functions whose names start with **`spawn`** and **`popen`** offer intermediate simplicity and power: they are cross-platform and not quite as simple as `system()`, but simple and usable enough for most purposes.

- The `exec` and `spawn` functions run a specified executable file given the executable file's path, arguments to pass to it, and optionally an environment mapping.

- The `system` and `popen` functions execute a *command*, a string passed to a new instance of the platform's default shell (typically /bin/sh on Unix). A *command* is a more general concept than an executable file, as it can include shell functionality (pipes, redirection, built-in shell commands).

IV.2.4.1 The exec functions

```
execl(path,*args)
execle(path,*args)
execlp(path,*args)
execv(path,args)
execve(path,args,env)
execvp(path,args)
execvpe(path,args,env)
```

- These functions run the executable file indicated by string *`path`*, replacing the current program (i.e., the Python interpreter) in the current process.

- `execlp`, `execvp`, and `execvpe` can accept a path argument that is just a filename rather than a complete path. In this case, the functions search for an executable file of that name along the directories listed in `os.environ['PATH']`.
  The other functions require path to be a complete path to the executable file for the new program.

- Functions whose names start with `execv` take a single argument `args` that is the sequence of the arguments to use for the new program.

- Functions whose names start with `execl` take the new program's arguments as separate arguments.

- `execle`, `execve`, and `execvpe` take an argument `env` that is a mapping to be used as the new program's environment, while the other functions use `os.environ` for this purpose.

- Each `exec` function uses the first item in `args` as the name under which the new program is told it's running; `args[1:]` are passed as arguments proper to the new program.

IV.2.4.2 The popen functions

- These functions are deprecated since Python 2.6, the **`subprocess`** module should be used instead.

`popen(cmd,mode='r',bufsize=-1)`

- Runs the string command *`cmd`* in a new process P, and returns a file-like object *`f`* that wraps a pipe to P's standard input or from P's standard output (depending on mode).
- When mode is '`r`' (or '`rb`', for binary-mode reading), `f` is read-only and wraps P's standard output.
  When mode is '`w`' (or '`wb`', for binary-mode writing), `f` is write-only and wraps P's standard input.
- The method `f.close()` waits for P to terminate, and returns `None` when P's termination is successful. However, if P's termination was unsuccessful, `f.close()` also returns the exit code.

`popen2(cmd,mode='t',bufsize=-1)`
`popen3(cmd,mode='t',bufsize=-1)`
`popen4(cmd,mode='t',bufsize=-1)`

- Each of these functions runs the string command *`cmd`* in a new process P, and returns a tuple of file-like objects that wrap pipes to P's standard input and from P's standard output and standard error.
- *`mode`* must be '`t`' to get file-like objects in text mode, or '`b`' to get them in binary mode.
- `bufsize` is an integer that denotes what buffering you request for the file objects:
  < 0, the operating system's default is used.
  0, the file is unbuffered.
  1, the file is line-buffered.
  > 1, the file uses a buffer of about bufsize bytes.
- `popen2` returns a pair `(fi,fo)`, where `fi` wraps P's standard input and `fo` wraps P's standard output.
- `popen3` returns a tuple with three items (`fi`, `fo`, `fe`), where `fe` wraps P's standard error.
- `popen4` returns a pair (`fi, foe`), where `foe` wraps both P's standard output and error.

Note: there is no way in which the caller of `popen2()`, `popen3()`, or `popen4()` can learn about P's termination code.

Note: Functions in the `popen` group are generally not suitable for driving another process interactively (i.e., writing something, then reading `cmd`'s response to that, then writing something else, and so on). The first time your program tries to read the response, if `cmd` is following a typical buffering strategy, everything blocks.

IV.2.4.3 The spawn functions

`spawnv(mode,path,args)`
`spawnve(mode,path,args,env)`

- These functions run the program indicated by *`path`* in a new process P, with the arguments passed as sequence *`args`*.

- `spawnve` uses mapping `env` as P's environment, while `spawnv` uses `os.environ` for this purpose.

- *`mode`* must be one of two attributes supplied by the `os` module:
    1. **`os.P_WAIT`** indicates that the calling process waits until the new process terminates,
    2. **`os.P_NOWAIT`** indicates that the calling process continues executing simultaneously with the new process.

- When mode is `os.P_WAIT`, the function returns the termination of P:
  0 indicates successful termination,
  c < 0 indicates P was killed by a signal,
  c > 0 indicates normal but unsuccessful termination.

- When mode is `os.P_NOWAIT`, the function returns P's process ID.

IV.2.4.4 The system function

`system(cmd)`

- Runs the string command *`cmd`* in a new process, and returns 0 if the new process terminates successfully.
  If the new process terminates unsuccessfully, `system()` returns an integer error code not equal to 0.

IV.2.5 Process Environment

- The operating system supplies each process with an environment, which is a set of **environment variables** whose names are identifiers and whose contents are strings.

- Module `os` supplies attribute **`environ`**, a mapping that represents the current process's environment.
  `os.environ` is initialized from the process environment when Python starts.

- Changes to `os.environ` update the current process's environment. These changes only affect the process itself, not others.

- Keys and values in `os.environ` must be strings.

```
>>> print(os.environ['HOME'])
/users/peter
```

IV.3 The commands module

- This module is deprecated since Python 2.6 and several methods have been removed in Python 3.X.

- Under Unix/Linux, the **`commands`** module contains wrapper functions for **`os.popen()`** which take a system command as a string and return any output generated by the command and, optionally, the exit status.

- The commands module defines the following functions:

`getstatusoutput(cmd)`

- Execute the string cmd in a shell with `os.popen()` and return a 2-tuple (status, output).

- The returned *`output`* will contain output and error messages.

`getoutput(cmd)`

- This functions is similar to `getstatusoutput()`, except that the exit status is ignored and the return value is a string containing the command's output.

`getstatus(file)`

- Return the output of "*`ls -ld file`*" as a string.

IV.4 The subprocess module

- The **subprocess** module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

- This module intends to replace several other, older modules and functions (`os.system`, `os.spawn*`, `os.popen*`, …).

IV.4.1 Using the subprocess Module

- This module defines one class called **Popen**. The constructor of this class may take up to 14 arguments … hopefully only a few of them are most of the time given:

  args should be a string, or a sequence of program arguments. The program to execute is normally the first item in the *args* sequence or the string if a string is given, but can also be explicitly set by using the ***executable*** argument.

  On Unix, with ***shell***`=False` (default) the `Popen` class uses `os.execvp()` to execute the child program.

  On Unix, with ***shell***`=True` the command is executed through the shell. The *executable* argument specifies which shell to use. On Unix, the default shell is /bin/sh.

  On Windows `Popen` uses `CreateProcess()` to execute the child program, which operates on strings. If *args* is a sequence, it will be converted to a string.

  bufsize, if given, has the same meaning as the corresponding argument to the built-in *open()* function: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size. The default value for *bufsize* is 0 (unbuffered).

  stdin, ***stdout*** and ***stderr*** specify the executed programs' standard input, standard output and standard error file handles, respectively. Valid values are ***PIPE***, an existing file descriptor (a positive integer), an existing file object, and `None`. ***PIPE*** indicates that a new pipe to the child should be created. With None, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, stderr can be ***STDOUT***, which indicates that the stderr data from the applications should be captured into the same file handle as for stdout.

  If ***cwd*** is not `None`, the child's current directory will be changed to *cwd* before it is executed. Note that this directory is not considered when searching the executable, so you can't specify the program's path relative to *cwd*.

  If ***env*** is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of inheriting the current process' environment, which is the default behaviour.

If ***`universal_newlines`*** is `True`, the file objects `stdout` and `stderr` are opened as text files, but lines may be terminated by any end-of-line convention, and are seen as '\n' by the Python program.

Note: Unlike some other popen functions, this implementation will never call /bin/sh implicitly. This means that all characters, including shell metacharacters, can safely be passed to child processes.

- Special Values:

  `PIPE`

  This special value that can be used as the *`stdin`*, *`stdout`* or *`stderr`* argument to `Popen` and indicates that a pipe to the standard stream should be opened.

  `STDOUT`

  Special value that can be used as the *`stderr`* argument to **`Popen`** and indicates that standard error should go into the same handle as standard output.

IV.4.2 Convenience Functions

- This module also defines two shortcut functions:

  `call(*popenargs, **kwargs)`

  Run command with arguments. Wait for command to complete, then return the *`returncode`* attribute.

  The arguments are the same as for the `Popen` constructor.

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode,
file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

  `check_call(*popenargs, **kwargs)`

  Run command with arguments. Wait for command to complete. If the exit code was zero then return, otherwise raise `CalledProcessError`. The `CalledProcessError` object will have the return code in the *`returncode`* attribute.

  The arguments are the same as for the `Popen` constructor.

IV.4.3 Exceptions

- Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent. Additionally, the exception object will have one extra attribute called child_traceback, which is a string containing traceback information from the childs point of view.

- The most common exception raised is `OSError`.

- A `ValueError` will be raised if `Popen` is called with invalid arguments.

- `check_call()` will raise `CalledProcessError`, if the called process returns a non-zero return code.

IV.4.4 The Popen Objects

- Instances of the `Popen` class have the following methods:

  `poll()`

  Check if child process has terminated. Set and return *`returncode`* attribute.

  `wait()`

  Wait for child process to terminate. Set and return *`returncode`* attribute.

  `communicate(input=None)`

  Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional *`input`* argument should be a string to be sent to the child process, or `None`, if no data should be sent to the child.

  `communicate()` returns a tuple (stdoutdata, stderrdata).

  Note that if you want to send data to the process's stdin, you need to create the `Popen` object with *`stdin=PIPE`*. Similarly, to get anything other than `None` in the result tuple, you need to give *`stdout=PIPE`* and/or *`stderr=PIPE`* too.

  `send_signal(signal)`

  Sends the signal signal to the child.

  `terminate()`

  Stop the child. On Posix OSs the method sends SIGTERM to the child. On Windows the Win32 API function `TerminateProcess()` is called to stop the child.

  `kill()`

  Kills the child. On Posix OSs the function sends SIGKILL to the child. On Windows `kill()` is an alias for `terminate()`.

- The following attributes are also available:

  `Warning`

  Use `communicate()` rather than *`.stdin.write`*, *`.stdout.read`* or *`.stderr.read`* to avoid deadlocks due to any of the other OS pipe buffers filling up and blocking the child process.

  `stdin`

  If the *`stdin`* argument was *`PIPE`*, this attribute is a file object that provides input to the child process. Otherwise, it is `None`.

  `stdout`

  If the *`stdout`* argument was *`PIPE`*, this attribute is a file object that provides output from the child process. Otherwise, it is `None`.

  `stderr`

  If the *`stderr`* argument was *`PIPE`*, this attribute is a file object that provides error output from the child process. Otherwise, it is `None`.

  `pid`

  The process ID of the child process.

  `returncode`

  The child return code, set by `poll()` and `wait()` (and indirectly by `communicate()`). A `None` value indicates that the process hasn't terminated yet.

  A negative value -N indicates that the child was terminated by signal N (Unix only).

IV.4.5 How to use the subprocess modules as a replacement of older functions

```
output=`mycmd myarg`
<=>
output = Popen(["mycmd", "myarg"],
          stdout=PIPE).communicate()[0]
```

```
output=`dmesg | grep hda`
<=>
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
output = p2.communicate()[0]
```

```
sts = os.system("mycmd" + " myarg")
<=>
p = Popen("mycmd" + " myarg", shell=True)
sts = os.waitpid(p.pid, 0)
```

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
<=>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

```
pipe = os.popen("cmd", 'r', bufsize)
<=>
pipe = Popen("cmd", shell=True, bufsize=bufsize,
          stdout=PIPE).stdout
```

```
pipe = os.popen("cmd", 'w', bufsize)
<=>
pipe = Popen("cmd", shell=True, bufsize=bufsize,
          stdin=PIPE).stdin
```

```
(child_stdin, child_stdout) = os.popen2("cmd", mode, bufsize)
<=>
p = Popen("cmd", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
```

V THREADING IN PYTHON

V.1 What Are Threads?

V.1.1 Processes

- Modern operating systems use timesharing to manage multiple programs which appear to the user to be running simultaneously.

- Assuming a standard machine with only one CPU, that simultaneity is only an illusion, since only one program can run at a time.

- Each program that is running counts as a process in UNIX terminology.

- The processes "take turns" running. One process might run for a few milliseconds, then be interrupted by the timer hardware, causing the OS to run. The OS saves the current state of the interrupted process so it can be resumed later, then selects the next process to give a turn to. This is known as a **context switch**; the context in which the CPU is running has switched from one process to another.

- This cycle repeats. Any given process will keep getting turns, and eventually will finish. A turn is called a ***timeslice***.

- The OS maintains a process table, listing all current processes. Each process will be shown as currently being in either *Run* state or *Sleep* state.

- Being in *Sleep* state means that the process is waiting for some event to occur (some user input for instance); we say that the process is blocked.

- Being in *Run* state does not mean that the process is currently running. It merely means that this process is ready to run, i.e. eligible for a turn.

V.1.2 Threads

- A ***thread*** is like a UNIX process. In fact, threads are sometimes called "lightweight" processes, because threads occupy much less memory, and take less time to create, than do processes.

- A major difference between processes and threads is that although each thread has its own local variables, just as is the case for a process, the global variables of the program from which the thread are being created are shared by all threads, and serve as the main method of communication between the threads.

V.1.3 Thread Managers

- The ***thread manager*** acts like a “mini-operating system” that is in charge of managing the threads created by a Python program.

- Just like a real OS maintains a table of processes, the thread manager maintains a table of threads. When one thread gives up the CPU, or has its turn pre-empted, the thread manager looks in the table for another thread to activate.

- Whichever thread is activated will then resume execution at the line at which it had left off.

- Just as a process is either in Run state or Sleep state, the same is true for a thread.

- In C or C++, the thread manager is in a library (`pthread` for instance), which is linked to the application program.

- In Python, the threads manager is in the interpreter itself. The manager may in turn call a more basic C/C++ threads library, making the picture more complex. This means that threads behaviour in Python may in fact depend on the underlying platform.

- The Python Thread Manager keeps track of how long the current thread has executing, in terms of the number of Python byte code instructions have executed.

- When that reaches a certain number, by default 100, another thread will be given a turn. We say that the thread has been ***pre-empted***. Such a switch will also occur if a thread reaches an I/O statement.

Note: the python documentation stated that “not all built-in functions that may block waiting for I/O allow other threads to run”.

V.1.4 The Global Interpreter Lock (GIL)

- Python maintains a Global Interpreter Lock (GIL) to ensure that only one thread has access to any Python object in a program at a time.

- This global lock must be held by the current thread before it can safely access Python objects.

  Without the lock competing threads could cause havoc, for example: when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

- Among other things, the existence of the GIL means that one cannot actually achieve true parallelism when one uses Python threading on a multiprocessor machine.

- In order to support multi threaded Python programs the interpreter regularly releases and reacquires the lock, by default every 100 bytecode instructions. This can however be changed using the **`sys.setcheckinterval()`** function. The lock is also released and reacquired around potentially blocking I/O operations like reading or writing a file, so that other threads can run while the thread that requests the I/O is waiting for the I/O operation to complete.

- To summarize, here is how the Python Virtual Machine executes in multi-threaded environment:

1. Set the GIL

2. Switch in a thread to run

3. Execute either ...

For a specified number of bytecode instructions, or
until the thread voluntarily yields control (can be accomplished using
`time.sleep(0.01)`), or
until the thread executes an I/O statements

4. Put the thread back to sleep (switch out thread)

5. Unlock the GIL, and ...

6. Do it all over again

Note 1: While `sys.setcheckinterval()` allows you to change the balance between responsiveness and efficiency (reducing the number of bytecodes increases responsiveness; increasing the number of bytecodes raises efficiency), the amount of time required to process each bytecode is too variable and unpredictable for this to be reliable.
Much more effective is to run Python in optimized mode (use `python -O` or set the **`PYTHONOPTIMIZE`** environment variable) to reduce the number of bytecodes

Note 2: The GIL only restricts pure Python code. Extensions (external Python libraries usually written in C) can be written that release the lock, which then allows the Python interpreter to run separately from the extension until the extension reacquires the lock. All of the standard Python blocking I/O calls are written this way.

V.1.5 Why Threading in Python?

- Threads (may) increase the responsiveness of your programs.

- Threads (may) also increase efficiency and speed of a program, not to mention the algorithmic simplicity.

V.2 The Basics

- Threading is supported via the **`_thread`** and **`threading`** modules. These modules are supposed to be optional, but most OS do support threading.

- The `_thread` module provides basic thread and locking support, while `threading` provides higher-level, fully featured thread management.

- It is recommended not to use the `_thread` module for many reasons:
  the high-level `threading` module is more contemporary,
  thread support in the `threading` module is much improved
  the use of attributes of the `_thread` module may conflict with using the `threading` module. the lower-level `_thread` module has only one synchronization primitives while `threading` has many,
  threads created thanks the `_thread` module are daemon threads (see VIII.4.1.2)
  …

- However, let us first see how to start a simple thread using the `thread` module. The code given below runs a simple thread in the background.

- You start a new thread using the **`start_new_thread()`** function which takes the address of the function that holds the statements to be run, along with arguments (passed in a `tuple`) to be passed to the function.

```
import time
import _thread

def myfunction(message,sleeptime,*args):
    while 1:
        print(message)
        time.sleep(sleeptime)  #sleep for a specified amount of time.

_thread.start_new_thread(myfunction,("Thread No:1",2))
while 1:pass
```

- A few other function are provided in the thread module, for instance:

  `interrupt_main`(   ) raise a `KeyboardInterrupt` exception in the main thread. A subthread can use this function to interrupt the main thread.

  `exit`() raise the `SystemExit` exception. When not caught, this will cause the thread to exit silently.

Note: There is a "main thread" object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread (see VIII.4.1.2)

V.3 Locks

- To make several threads running is as simple as calling `start_new_thread()` multiple times.

- Because all the running threads share the same memory, you must make sure they don't mess up the variables for each other, or try to modify the same things at the same time, creating a mess. These issues fall under the heading of ***synchronization***.

- With the `thread` module, synchronization is done using a lock (a **`LockType`** object). Locks are created using the **`allocate_lock()`** factory function.

Note: a lock can also be created thanks to the factory function **`Lock()`** of the `threading` module.

- Locks are used as ***mutex*** objects, and are used for handling critical sections of code.

- A thread enters the critical section by calling the **`acquire(`*`[waitflag]`*`)`** method, which can either be blocking or non-blocking. A thread exits the critical section, by calling the **`release()`** method.

- When the state is locked, `acquire()` blocks until a call to `release()` in another thread changes it to unlocked, then the `acquire()` call resets it to locked and returns.

Note: if `acquire()` is called with an integer argument 0, it does not block even if the lock is not available, in that case, it simply return *`False`* immediately (if the argument is nonzero, the lock is acquired unconditionally as without argument).

- When more than one thread is blocked in `acquire()` waiting for the state to turn to unlocked, only one thread proceeds when a `release()` call resets the state to unlocked.

- The **`locked()`** method returns the status of the lock: `True` if it has been acquired by some thread, `False` if not.

- The following listing shows how to use the `Lock` object.

```
import time
import _thread

def myfunction(string,sleeptime,lock,*args):
    while 1: #entering critical section
        lock.acquire()
        print(string," Now Sleeping after Lock acquired for",\
 sleeptime)
        time.sleep(sleeptime)
        print(string," Now releasing lock and then sleeping\
 again")
        lock.release()
    #exiting critical section

lock=_thread.allocate_lock()
_thread.start_new_thread(myfunction,("Thread No:1",2,lock))
_thread.start_new_thread(myfunction,("Thread No:2",2,lock))

while 1:pass
```

- The code given above is fairly straightforward. We call `lock.acquire()` just before entering the critical section and then call `lock.release()` to exit the critical section.

- The output of the above example is something like:

```
Thread No:2  Now Sleeping after Lock acquired for  2
Thread No:2  Now releasing lock and then sleeping again
Thread No:1  Now Sleeping after Lock acquired for  2
Thread No:1  Now releasing lock and then sleeping again
Thread No:2  Now Sleeping after Lock acquired for  2
Thread No:2  Now releasing lock and then sleeping again
Thread No:1  Now Sleeping after Lock acquired for  2
Thread No:1  Now releasing lock and then sleeping again
Thread No:2  Now Sleeping after Lock acquired for  2
...
```

- In addition to these methods, lock objects can also be used via the `with` statement:

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

V.4 Using the threading Module

- The **`threading`** module uses the built in **`thread`** module to provide some very interesting features that would make your programming a whole lot easier.

- There are in built mechanisms, which provide critical section locks, wait/notify locks, etc.

- The major components of the `Threading` module are:

  `Lock` , **`RLock`** , **`Semaphore`** , **`Condition`** , **`Event`** , **`Thread`** , **`Local`** and **`Timer`**

V.4.1 The Thread class

- The **`Thread`** class is a wrapper to the `start_new_thread()` function, which we saw earlier, but with a little more functionality.

- There are two ways you can create threads using the `Thread` class:

  1. Create `Thread` instance, passing a callable instance to the constructor
  2. Subclass `Thread` and override the `run()` method

V.4.1.1 Creation of a Thread

- The `Thread` class contains a member method `run()`. This is an empty method, to be overridden with the application-specific function to be run by the thread.

- If you've passed a callable as argument to the constructor of the class Thread, it will be used instead of the `run()` method.

- Executing the thread is simple. All you have to do is create an instance of the thread class and then call its **`start()`** method.

- Once the thread's activity is started, the thread is considered 'alive'. It stops being alive when its `run()` method terminates. The **`threading.isAlive()`** method tests whether the thread is alive.

- If you want to pass initial arguments when creating the thread, you will need to override the `Thread` class's **`__init__()`** method to accept parameters.
  If the subclass overrides the constructor, it must make sure to invoke the base class constructor (`Thread.__init__()`) before doing anything else to the thread.

```
import time
import threading

class thread1 (threading.Thread):
     def __init__(self, nb):
        threading.Thread.__init__(self)
        self.nb=nb
     def run(self):
        while 1:
            print("Loop", self.nb)
            time.sleep(1)

t=thread1(56)
t.start()
t.join()
```

- Compared to the previous example (that uses the `thread` module), this program ends in a different way:

```
t.join()
```

- The **`join()`** method in the class `Thread` blocks the calling thread until the thread whose `join()` method is called terminates - either normally or through an unhandled exception - or until an optional timeout occurs.

- In the previous example, the overall effect of this statement is that the "main" program will wait at that point the thread *`t`* is done.

- This is a much cleaner approach than what we used earlier, and it is also more efficient, since the script will not be given any turns in which it wastes time looping around doing nothing, as in the previous program.

- You can get a list of the currently started threads via the function **`threading.enumerate()`**.

- You can get the number of `Thread` objects currently started via the function **`threading.activeCount()`**.

- The **`threading.current_thread()`** function returns the id of the currently running thread.

- The **`threading.settrace(`*`func`*`)`** function set a trace function for all threads started from the `threading` module. The *func* will be passed to `sys.settrace()` for each thread, before its `run()` method is called.

- The **`threading.profile(`*`func`*`)`** function set a profile function for all threads started from the `threading` module. The *func* will be passed to `sys.profile()` for each thread, before its `run()` method is called.

V.4.1.2 Daemon Thread

- Daemon threads are, generally, service providers for other threads.
- Any thread can be a daemon thread.
- To make a thread be a daemon thread, you must call the **`setDaemon()`** method with the parameter `True` (this must be done before the thread is started).
- To determine if a thread is a daemon, you can use the **`isDaemon()`** method.
- When the only remaining threads in a process are daemon threads, the Python interpreter exits. If you prefer, the entire Python program exits when no active non-daemon threads are left.

V.4.1.3 Other methods

- You can give a name to a thread using the **`setName()`** method and retrieve this name using the **`getName()`** method. The name of the thread can also be given at the level of the thread constructor.

V.4.1.4 Threads limitations

- There are no priorities and no thread groups.
- Threads do not have methods to stop, suspend/resume or interrupt them.

V.4.2 Lock and RLock

- The factory function **`threading.Lock()`** can be used to create a Lock instance. The facilty offered by such a lock are similar to the one offered in the thread module.
- **`Rlock`** (or reentrant lock) provides a mechanism for a thread to acquire multiple instances of the same lock, each time incrementing the depth of locking when acquiring and decrementing the depth of locking when releasing.
- `RLock` makes it very easy to write code which conforms to the classical Readers/Writers Problem.
- A `RLock` is created thanks to the factory function **`threading.RLock()`**.
- Like a normal lock, if a `RLock` is in the locked state, some thread owns the lock; in the unlocked state, no thread owns it.
- To lock the lock, a thread calls its `acquire()` method; this returns once the thread owns the lock.
- To unlock the lock, a thread calls its `release()` method.
- `acquire()/release()` call pairs may be nested; only the final `release()` (the `release()` of the outermost pair) resets the lock to unlocked and allows another thread blocked in `acquire()` to proceed.

V.4.3 Condition variables

- Condition variables are a way of synchronizing access between multiple threads, which wait for a particular condition to be true to start any major processing.

- At the first level, there is the class **`threading.Condition`**, which can be used to create condition variables. However, at this level condition variables are rather cumbersome to use, as not only do we need to set up condition variables but we also need to set up locks to guard them.

- Python offers a higher-level class, **`threading.Event`**, which is just a wrapper for **`threading.Condition`**, but does all the lock operations behind the scenes, alleviating the programmer from this work.

- A condition variable constructor takes a lock object as argument or if none is supplied creates its own `RLock` object.

- A condition variable has `acquire()` and `release()` methods that call the corresponding methods of the associated lock.

- A Thread waits for a particular condition to be true by using the **`wait()`** method, while it signals another thread by using the **`notify()`** or **`notifyAll()`** method.

- These three methods must only be called when the calling thread has acquired the lock using the now well known `acquire()` and `release()` methods..

- The `wait()` method releases the lock, and then blocks until it is awakened by a `notify()` or `notifyAll()` call for the same condition variable in another thread. Once awakened, it re-acquires the lock and returns.

- The **`wait()`** method has an optional argument specifying the number of seconds it should wait before it times out.

- The **`notify()`** method wakes up one of the threads waiting for the condition variable, if any are waiting.

- The **`notifyAll()`** method wakes up all threads waiting for the condition variable.

Note: the `notify()` and `notifyAll()` methods don't release the lock.

- The typical programming style using condition variables uses the lock to synchronize access to some shared data (a container for instance); threads that are interested in a particular change of state (the container is not empty, or the container is not full, for instance) call `wait()` repeatedly until they see the desired state, while threads that modify the state (add or remove an element, for instance) call `notify()` or `notifyAll()` when they change the state in such a way that it could possibly be a desired state for one of the waiters.

```
import thread
import time
from threading import *

class itemQ:
    def __init__(self):
        self.count=0
    def produce(self,num=1):
        self.count+=num
    def consume(self):
        if self.count: self.count-=1
    def isEmpty(self):
        return not self.count

class Producer(Thread):
    def __init__(self,condition,itemq,sleeptime=1):
        Thread.__init__(self)
        self.cond=condition
        self.itemq=itemq
        self.sleeptime=sleeptime
    def run(self):
        cond=self.cond
        itemq=self.itemq
        while 1 :
            cond.acquire() #acquire the lock
            print(current_thread(),"Produced One Item")
            itemq.produce()
            cond.notifyAll()
            cond.release()
            time.sleep(self.sleeptime)

class Consumer(Thread):
    def __init__(self,condition,itemq,sleeptime=2):
        Thread.__init__(self)
        self.cond=condition
        self.itemq=itemq
        self.sleeptime=sleeptime
    def run(self):
        cond=self.cond
        itemq=self.itemq
        while 1:
            time.sleep(self.sleeptime)
            cond.acquire() #acquire the lock

            while itemq.isEmpty():
                cond.wait()

            itemq.consume()
            print(current_thread(),"Consumed One Item")
            cond.release()
```

```
if __name__=="__main__":

    q=itemQ()

    cond=Condition()

    pro=Producer(cond,q)
    cons1=Consumer(cond,q)
    cons2=Consumer(cond,q)

    pro.start()
    cons1.start()
    cons2.start()
    while 1: pass
```

V.4.3 Semaphore and bounded semaphore

- A ***semaphore*** manages an internal counter which is decremented by each `acquire()` call and incremented by each `release()` call.

- The factory method **`threading.Semaphore(`*`[value=1]`*`)`** can be used to create a semaphore.

- A ***bounded semaphore*** behaves like a semaphore except that its make sure its counter current value doesn't exceed its initial value. If it does, a `ValueError` exception is raised.

- The factory method **`threading.BoundedSemaphore(`*`[value=1]`*`)`** can be used to create a bounded semaphore.

- The counter can never go below zero; when `acquire()` finds that it is zero, it blocks, waiting until some other thread calls `release()`.

- The `Semaphore()` and `BoundedSemaphore()` method take an optional argument that represents the initial value for the internal counter; it defaults to 1.

- Bounded semaphores are often used to guard resources with limited capacity: a pool of database connections for instance. The semaphore counter is initialised with the number of connections available in the pool, and then, each time a thread need a database connection, it calls the `acquire()` method which decrements the counter. The thread will automatically be blocked if the semaphore counter reaches the value 0. When the connection is closed, the thread will call the `release()` method which increments the counter.

V.4.4 Event

- The **`Event`** is actually a thin wrapper on the **Condition variable** so that we don't have to mess about with locks.

- An `Event` is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.

- The factory method **`threading.Event()`** can be used to create an `Event` object.

- An `Event` object manages an internal flag that can be set to true with the **`set()`** method and reset to false with the **`clear()`** method.

- The **`wait([timeout])`** method blocks until the flag is true.

- The **`isSet()`** methods tests whether the flag is set or not.

V.4.5 Timer

- The **`threading.Timer`** class represents an action that should be run only after a certain amount of time has passed.
- `Timer` is a subclass of `Thread`.
- Timers are started, as with threads, by calling their **`start()`** method. The timer can be stopped (before its action has begun) by calling the **`cancel()`** method.
- The interval the timer will wait before executing is given (in seconds) as the first argument of the `Timer()` constructor. The second argument of the constructor is the name of a function that will be executed when the time has elapsed. An optional third argument will correspond to argument passed to the function when it is activated.

```
import threading
def hello():
    print("hello, world")

t = threading.Timer(30.0, hello)
t.start()                       # after 30 seconds, "hello, world" will be printed
```

V.4.6 Local data

- The **`threading.Local`** class represents thread-local data. Thread-local data are data whose values are thread specific.
- To manage thread-local data, just create an instance of local (or a subclass) and store attributes on it:

```
mydata = threading.local()
mydata.x = 1
```

- The instance's values will be different for separate threads.

V.5 Queue

- Queues are FIFO containers provided by the **`Queue`** module.
- Queues provide a simple and efficient way of implementing stack, priority queue etc.
- Queues are thread-safe, they handle both data protection and synchronization.
- The methods offered by the **`Queue`** class are:

  `Queue(`*`maxsize`*`)` : (the constructor) *maxsize* is an integer that sets the maximum number of items that can be placed in the queue. Insertion will block once this size has been reached. If *maxsize* is less than or equal to zero, the queue size is infinite.

  `put(item)`, **`put_nowait(item)`** : to put *item* into the queue.

  `get()` , **`get_nowait()`** : Remove and return an item from the queue

  `qsize()` : Return the size of the queue

  `empty()` , **`full()`**: Return True if the queue is empty or full respectively.

VI THE MULTIPROCESSING MODULE

- The multiprocessing module includes a relatively simple API for dividing work up between multiple processes. It is based on the API for *threading*, and in some cases is a drop-in replacement.

- Features provided by multiprocessing but not available in threading are covered later.

VI.1 Multiprocessing Basics

- The simplest way to spawn a second process is to instantiate a **`Process`** object with a target function and call **`start()`** to let it begin working.

```
import multiprocessing

def worker():
    """worker function"""
    print('Worker')
    return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker)
        jobs.append(p)
        p.start()
```

- It usually more useful to be able to spawn a process with arguments to tell it what work to do. Unlike with threading, to pass arguments to a multiprocessing `Process` the argument must be able to be serialized using **`pickle`**.

```
p = multiprocessing.Process(target=worker, args=(i,))
```

VI.1.1 Importable Target Functions

- One difference between the threading and multiprocessing examples is the extra protection for `__main__` used in the multiprocessing examples. Due to the way the new processes are started, the child process needs to be able to import the script containing the target function.

- Wrapping the main part of the application in a check for `__main__` ensures that it is not run recursively in each child as the module is imported. Another approach is to import the target function from a separate script.

VI.1.2 Determining the Current Process

- Passing arguments to identify or name the process is cumbersome, and unnecessary. Each `Process` instance has a name with a default value that can be changed as the process is created. Naming processes is useful for keeping track of them, especially in applications with multiple types of processes running simultaneously.

```
worker_1 = multiprocessing.Process(name='worker 1',
target=worker)

name = multiprocessing.current_process().name
```

VI.1.3 Daemon Processes

- By default the main program will not exit until all of the children processes have exited. There are times when starting a background process that runs without blocking the main program from exiting is useful

- To mark a process as a *daemon*, set its **`daemon`** attribute with a `True` value. The default is for processes to not be daemons, so passing `True` turns the daemon mode on.

```
    d = multiprocessing.Process(name='daemon', target=daemon)
    d.daemon = True
```

- The daemon process is terminated automatically before the main program exits, to avoid leaving orphaned processes running.

VI.1.4 Waiting for Processes

- To wait until a process has completed its work and exited, use the **`join()`** method.

```
n = multiprocessing.Process(name='daemon', target=non_daemon)
n.daemon = True
n.start()
time.sleep(1)
n.join()
```

- By default, `join()` blocks indefinitely. It is also possible to pass a timeout argument (a float representing the number of seconds to wait for the process to become inactive). If the process does not complete within the timeout period, `join()` returns anyway.

VI.1.5 Terminating Processes

- Although it is better to use the “poison pill” method of signalling to a process that it should exit, if a process appears hung or deadlocked it can be useful to be able to kill it forcibly. Calling **`terminate()`** on a process object kills the child process.

```
p = multiprocessing.Process(target=slow_worker)
print 'BEFORE:', p, p.is_alive()

p.start()
print 'DURING:', p, p.is_alive()

p.terminate()
print 'TERMINATED:', p, p.is_alive()

p.join()
print 'JOINED:', p, p.is_alive()
```

Note: It is important to `join()` the process after terminating it in order to give the background machinery time to update the status of the object to reflect the termination.

VI.1.6 Process Exit Status

- The status code produced when the process exits can be accessed via the **`exitcode`** attribute. The `exitcode` values can be:

0 :	no error was produced
> 0 :	the process had an error, and exited with that code
< 0 :	the process was killed with a signal of -1 * exitcode

VI.1.7 Logging

- When debugging concurrency issues, it can be useful to have access to the internals of the objects provided by multiprocessing. There is a convenient module-level function to enable logging called **`log_to_stderr()`**. It sets up a `logger` object using `logging` and adds a handler so that log messages are sent to the standard error channel.

```
import multiprocessing
import logging
import sys

def worker():
    print('Doing some work')
    sys.stdout.flush()

if __name__ == '__main__':
    multiprocessing.log_to_stderr(logging.DEBUG)
    p = multiprocessing.Process(target=worker)
    p.start()
    p.join()
```

- By default the logging level is set to `NOTSET` so no messages are produced. Pass a different level to initialize the logger to the level of detail you want.

- To manipulate the logger directly (change its level setting or add handlers), use `get_logger()`.

```
    …
    multiprocessing.log_to_stderr()
    logger = multiprocessing.get_logger()
    logger.setLevel(logging.INFO)
    …
```

- The logger can also be configured through the logging configuration file API, using the name multiprocessing.

VI.1.8 Subclassing Process

- Although the simplest way to start a job in a separate process is to use Process and pass a target function, it is also possible to use a custom subclass.

- The derived class should override **`run()`** to do its work.

VI.2 Communication between Processes

- As with threads, a common use pattern for multiple processes is to divide a job up among several workers to run in parallel. Effective use of multiple processes usually requires some communication between them, so that work can be divided and results can be aggregated.

VI.2.1 Passing Messages to Processes

- A simple way to communicate between process with multiprocessing is to use a `Queue` to pass messages back and forth. You can create a `Queue` instance thnks to the `Queue()` constructor provided by the `multiprocessing` module. Any pickle-able object can pass through a `Queue`.

VI.2.2 Signalling between Processes

- The **`Event`** class provides a simple way to communicate state information between processes. An event can be toggled between set and unset states. Users of the event object can wait for it to change from unset to set, using an optional timeout value.

```
import multiprocessing
import time

def wait_for_event(e):
    """Wait for the event to be set before doing anything"""
    print('wait_for_event: starting')
    e.wait()
    print('wait_for_event: e.is_set()->', e.is_set())

def wait_for_event_timeout(e, t):
    """Wait t seconds and then timeout"""
    print('wait_for_event_timeout: starting')
    e.wait(t)
    print('wait_for_event_timeout: e.is_set()->', e.is_set())

if __name__ == '__main__':
    e = multiprocessing.Event()
    w1 = multiprocessing.Process(name='block',
                                 target=wait_for_event,
                                 args=(e,))
    w1.start()
    w2 = multiprocessing.Process(name='non-block',
                                target=wait_for_event_timeout,
                                args=(e, 2))
    w2.start()
    print('main: waiting before calling Event.set()')
    time.sleep(3)
    e.set()
    print('main: event is set')
```

- When `wait()` times out it returns without an error. The caller is responsible for checking the state of the event using `is_set()`.

```
$ python -u multiprocessing_event.py

main: waiting before calling Event.set()
wait_for_event: starting
wait_for_event_timeout: starting
wait_for_event_timeout: e.is_set()-> False
main: event is setwait_for_event: e.is_set()-> True
```

VI.2.3 Controlling Access to Resources

- In situations when a single resource needs to be shared between multiple processes, a **`Lock`** can be used to avoid conflicting accesses.

```
import multiprocessing
import sys

def worker_with(lock):
    with lock:
        sys.stdout.write('Lock acquired via with\n')

def worker_no_with(lock):
    lock.acquire()
    try:
        sys.stdout.write('Lock acquired directly\n')
    finally:
        lock.release()

if __name__=="__main__":
    lock = multiprocessing.Lock()
    out=open("data.out", "w")
    w = multiprocessing.Process(target=worker_with,
args=(lock, ))
    nw = multiprocessing.Process(target=worker_no_with,
args=(lock, ))

    w.start()
    nw.start()
```

- In this example, the messages printed to the console may be jumbled together if the two processes do not synchronize their access of the output stream with the lock.

Note: On Windows the subprocesses will import (i.e. execute) the main module at start. You need to protect the main code with if __name__=='__main__' ... to avoid creating subprocesses recursively.

VI.2.4 Synchronizing Operations

- Condition objects can be used to synchronize parts of a workflow so that some run in parallel but others run sequentially, even if they are in separate processes.

```
import multiprocessing
import time

def stage_1(cond):
    """perform first stage of work, then notify stage_2 to 
continue"""
    name = multiprocessing.current_process().name
    print ('Starting', name)
    with cond:
        print('{} done and ready for stage 2'.format(name))
        cond.notify_all()

def stage_2(cond):
    """wait for the condition telling us stage_1 is done"""
    name = multiprocessing.current_process().name
    print ('Starting', name)
    with cond:
        cond.wait()
        print ('{} running'.format(name))

if __name__ == '__main__':
    condition = multiprocessing.Condition()
    s1 = multiprocessing.Process(name='s1', target=stage_1, 
args=(condition,))
    s2_clients = [
        multiprocessing.Process(name='stage_2[{}]'.format(i), 
target=stage_2, args=(condition,))
        for i in range(1, 3)
        ]

    for c in s2_clients:
        c.start()
        time.sleep(1)
    s1.start()

    s1.join()
    for c in s2_clients:
        c.join()
```

- In this example, two process run the second stage of a job in parallel, but only after the first stage is done.

VI.2.5 Controlling Concurrent Access to Resources

- Sometimes it is useful to allow more than one worker access to a resource at a time, while still limiting the overall number. A `Semaphore` is one way to manage those connections.

```
import random
import multiprocessing
import time

class ActivePool(object):
    def __init__(self):
        super(ActivePool, self).__init__()
        self.mgr = multiprocessing.Manager()
        self.active = self.mgr.list()
        self.lock = multiprocessing.Lock()
    def makeActive(self, name):
        with self.lock:
            self.active.append(name)
    def makeInactive(self, name):
        with self.lock:
            self.active.remove(name)
    def __str__(self):
        with self.lock:
            return str(self.active)

def worker(s, pool):
    name = multiprocessing.current_process().name
    with s:
        pool.makeActive(name)
        print('Now running: {}'.format(str(pool)))
        time.sleep(random.random())
        pool.makeInactive(name)

if __name__ == '__main__':
    pool = ActivePool()
    s = multiprocessing.Semaphore(3)
    jobs = [
        multiprocessing.Process(target=worker, name=str(i),
args=(s, pool))
        for i in range(10)
        ]

    for j in jobs:
        j.start()

    for j in jobs:
        j.join()
        print('Now running: {}'.format(str(pool)))
```

- In this example, the `ActivePool` class simply serves as a convenient way to track which processes are running at a given moment. A real resource pool would probably allocate a connection or some other value to the newly active process, and reclaim the value when the task is done. Here, the pool is just used to hold the names of the active processes to show that only three are running concurrently.

VI.2.6 Managing Shared State

- In the previous example, the list of active processes is maintained centrally in the `ActivePool` instance via a special type of list object created by a `Manager`. The `Manager` is responsible for coordinating shared information state between all of its users.

```
import multiprocessing

def worker(d, key, value):
    d[key] = value

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    d = mgr.dict()
    jobs = [ multiprocessing.Process(target=worker, args=(d, 
i, i*2))
             for i in range(10)
             ]
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()
    print('Results:', d)
```

- By creating the list through the manager, it is shared and updates are seen in all processes. Dictionaries are also supported.

VI.3 Process Pools

- The **`Pool`** class can be used to manage a fixed number of workers for simple cases where the work to be done can be broken up and distributed between workers independently. The return values from the jobs are collected and returned as a list. The pool arguments include the number of processes and a function to run when starting the task process (invoked once per child).

```
import multiprocessing

def do_calculation(data):
    return data * 2

def start_process():
    print('Starting', multiprocessing.current_process().name)

if __name__ == '__main__':
    inputs = list(range(10))
    print('Input   :', inputs)

    builtin_outputs = map(do_calculation, inputs)
    print('Built-in:', builtin_outputs)

    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(processes=pool_size,
                                initializer=start_process,
                                )
    pool_outputs = pool.map(do_calculation, inputs)
    pool.close() # no more tasks
    pool.join()  # wrap up current tasks

    print('Pool    :', pool_outputs)
```

- The result of the `map()` method is functionally equivalent to the built-in `map()`, except that individual tasks run in parallel. Since the pool is processing its inputs in parallel, `close()` and `join()` can be used to synchronize the main process with the task processes to ensure proper cleanup.

- By default `Pool` creates a fixed number of worker processes and passes jobs to them until there are no more jobs. Setting the **`maxtasksperchild`** parameter tells the pool to restart a worker process after it has finished a few tasks. This can be used to avoid having long-running workers consume ever more system resources.

```
import multiprocessing

def do_calculation(data):
    return data * 2

def start_process():
    print('Starting', multiprocessing.current_process().name)

if __name__ == '__main__':
    inputs = list(range(10))
    print('Input   :', inputs)

    builtin_outputs = map(do_calculation, inputs)
    print('Built-in:', builtin_outputs)

    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(processes=pool_size,
                                initializer=start_process,
                                maxtasksperchild=2,
                                )
    pool_outputs = pool.map(do_calculation, inputs)
    pool.close() # no more tasks
    pool.join()  # wrap up current tasks

    print('Pool    :', pool_outputs)
```

- The pool restarts the workers when they have completed their allotted tasks, even if there is no more work. In this output, eight workers are created, even though there are only 10 tasks, and each worker can complete two of them at a time.

VII THE DATABASE API

- This chapter discusses the Python **DataBase API**, a standardized way of connecting to SQL databases.

- There are lots and lots of SQL databases available out there, and many of them have corresponding client modules in Python.

- Most of the basic functionality of all the databases is the same, but their interfaces (APIs) are different.

- In order to solve this problem for database modules in Python, a standard database API has been agreed upon: the DataBase API (2.0).

- Today, the Python client modules are, for most of them, compliant with this API. Such modules exist for Oracle, MS SQL, MySQL, Postgress, …

- For example, to access the MySQL database using Python 3.8, you can install the **`mysqlclient`** module using the following procedure:

  1. With a standard python distribution:
     `pip install` **`mysqlclient`**

  2. With anaconda:
     `conda install` **`mysqlclient`**

  3. Read the documentation:
     https://github.com/PyMySQL/mysqlclient-python/blob/master/doc/user_guide.rst
     http://zetcode.com/db/mysqlpython/

VII.1 Global Variables

- Any compliant (that is, compliant with the DB API, version 2.0) database module must have three global variables, which describe the peculiarities of the module.

- The API is designed to be very flexible, and to work with several different underlying mechanisms without too much wrapping. If you want your program to work with several different databases, this can be a nuisance, because you have to cover many different possibilities.

- A more realistic course of action, in many cases, would be to simply check these variables to see that a given database module is acceptable to your program.

- The global variables are:

`apilevel`	The version of the Python DB API in use
`threadsafety`	How thread-safe the module is
`paramstyle`	Which parameter style is used in the SQL queries

- The **`apilevel`** is simply a string constant, giving the API version in use ('1.0' or '2.0'). If the variable isn't there, the module is not 2.0-compliant.

- The **`threadsafety`** level is an integer ranging from 0 to 3, inclusive:

    - 0 means that threads may not share the module at all,
    - 1 means that threads may share the module itself, but not connections
    - 2 means that threads may share modules and connections, but not cursors
    - 3 means that the module is completely thread-safe.

- The **`paramstyle`** indicates how parameters are spliced into SQL queries when you make the database perform multiple similar queries (see VII.4.4)

VII.2 Exceptions

- The API defines several exceptions, to make fine-grained error handling possible.

- These exceptions defined in a hierarchy, so you can also catch several types of exceptions with a single except block.

- The exception hierarchy is shown in the following table.

`StandardError`	Generic superclass of all exceptions
`Warning`	a subclass of StandardError raised if a nonfatal problem occurs
`Error`	a subclass of StandardError which represents a superclass of all error conditions
`InterfaceError`	subclass of Error which represents Errors relating to the interface, not the database
`DatabaseError`	subclass of Error which represents Errors relating to the database
`...`	

VII.3 Opening the connection

- Before a database can be accessed, we need to open a connection and store the connection object in a variable.

- In the dbapi-2.0 paradigm, connections to the database are represented by **`connection`** objects, created by calling the **`connect()`** function giving it several arguments.

- These arguments are module-dependent (fortunately the rest of the dbapi-2.0 is module-independent) and the better thing you can do is to consult your favourite module documentation.

- It is possible to open any number of connections to different databases and, obviously, multiple connections to the same database.

- If the database supports transactions, each connection is guaranteed to be independent (from the transactional point of view) from other connections to the same database.

Note: A connection object does not map directly to a physical connection; even if most modules implement a one-to-one correspondence, some use pools of multiple physical connections to better support concurrent execution and threading.

VII.3.1 The connect() function

- The prototype for the **`connect()`** function call is given below.

  `connect(dsn, user, password, host, database, ...)`

- The DBAPI standard recommends, but does not mandate, that connect should at least accept optional arguments with the following names:

`database`	Name of the specific database to connect
`dsn`	Data Source Name, a module-dependent argument.
`host`	Hostname on which the database is running
`password`	Password to use for the connection
`user`	Username for the connection

- For instance, the `connect()` function of the MySQLdb module uses the following parameters: db, host, user, passwd as in the following example:

```
import MySQLdb

conn = MySQLdb.connect ( host = "localhost",
                         user = "testuser",
                         passwd = "testpass",
                         db = "test")
```

- To allow for the possibility of connection failure it is necessary to catch exceptions. The kind of exceptions that could be raised depends on the DB module you use.

- With the `MySQLdb` module, the resulting connection sequence looks like this:

```
import MySQLdb
import sys
try:
    conn = MySQLdb.connect (host = "localhost",
                           user = "alogname",
                           passwd = "apasswd",
                           db = "test")
except MySQLdb.Error as e:
    print("Error {}: {}".format(e.args[0], e.args[1]))
    sys.exit (1)
```

- The exception class `MySQLdb.Error` gives access to database-specific error information that MySQLdb can provide. If an exception occurs, MySQLdb makes this information available in `e.`*`args`*, a two-element tuple containing the numeric error code and a string describing the error.

- Any database-related statements should be placed in a similar `try/except` structure to trap and report errors.

VII.3.2 The Connection class

- The class Connection supplies the following methods:

`close()`

- Terminates the database connection and releases all related resources.

Note: you should call `close()` as soon as you're done with the database, since keeping database connections uselessly open can be a serious resource drain on the system.

`commit()`

- Commits the current transaction in the database.

`cursor()`

- Returns a new instance of class **`Cursor`**, covered later in this chapter.

`rollback()`

- Rolls back the current transaction in the database.

VII.3.3 Transactions

- The dbapi-2.0 document specifies that transactions should be worked out at connection level (when the underlying database provides transactions; if this is not the case all the transaction-related functions are no-ops.)

- If the database supports transactions, all the “work” done via a connection (i.e., all the changes done to the database) is not “saved” until you commit it to the database.

- You can do that by calling the **`commit()`** method on the connection object

- If your program encounters an error or ends up in an unusual situation and you don’t want to commit your changes to the database you can undo all the changes (from the last commit on) using the **`rollback()`** method.

Note: a relational database does not have unlimited undo levels, only one. Calling `rollback()` again and again or calling it after a `commit()` will simply not work.

VII.3.4 Closing the connection

- A connection can be explicitly closed by invoking the **`close()`** method on the connection object or by explicitly destroying the connection object using the `del` operator.

- An important point is that if the module supports the transaction model, closing a connection does an implicit rollback on it. That simply means that you’ll need to call `commit()` one last time before letting the connection die.

VII.4 Executing Queries

- Just as you need a `Connection` object to manage your database connections, you will use **`Cursor`** objects to manage the execution of queries and to retrieve data.

- Some databases (like PostgreSQL, Oracle and others) support cursors natively and have exotic functionality accessible only when using them. The dbapi-2.0 cursor objects share nothing with (real) database cursors, they are completely different stuff.

- The dbapi-2.0 defines a set of well-defined cursor methods and attributes and, even if almost every dbapi-2.0 compliant module extends the cursor interface somehow, the basics work the same for every module.

Note: Even if the method provided by `Cursor` are similar on all modules, the SQL code you send to the database can contain non-standard features or features not supported by every database backend.

- A cursor object is always associated with an open connection and is created by calling the **`cursor()`** method on the connection.

- The `cursor()` method takes no arguments and returns a new `Cursor` object.

- A new cursor is valid and can be used to send queries to the backend until the cursor itself or its parent connection are closed; after that trying to use the cursor shall raise an exception.

VII.4.1 The Cursor methods

- The Cursor class offers the following methods:

Name	Description
`callproc(name[, params])`	Call a named database procedure with given name and params (optional).
`close()`	Close the cursor. Cursor is now unusable.
`execute(oper[, params])`	Execute an SQL operation, possibly with parameters.
`executemany(oper, pseq)`	Execute an SQL operation for each param set in a sequence.
`fetchone()`	Fetch the next row of a query result set as a sequence, or None.
`fetchmany([size])`	Fetch several rows of a query result set. Default size is cursor.arraysize (1 by default)
`fetchall()`	Fetch all (remaining) rows as a sequence of sequences.
`nextset()`	Skip to the next available result set (optional).
`setoutputsize(size[, col])`	Set a buffer size for fetching big data values.

- The Cursor class offers the following variables:

Name	Description
`description`	Sequence of result column descriptions. Read-only.
`rowcount`	The number of rows in the result. Read-only.
`arraysize`	How many rows to return in fetchmany. Default is 1.

VII.4.2 The execute() method

- **`execute()`** is surely the most important cursor method and the only way to send SQL queries to the database.

- It takes one mandatory argument (the query string) and an optional list (or dictionary) of bound variables:

  `execute(query, vars=None)`

- `execute()` makes really easy to have the database execute literal SQL code, as in the following examples.

- First we drop the *animal* table if it already exists, to begin with a clean state, Then we create the *animal* table. Finally, we insert some data into the table and report the number of rows added. Each query is issued by invoking the cursor object's `execute()` method.

```
cursor = conn.cursor ()
cursor.execute ("DROP TABLE IF EXISTS animal")
cursor.execute (""" CREATE TABLE animal(
                       name CHAR(40),
                       category CHAR(40))""")
cursor.execute ("""
           INSERT INTO animal (name, category)
           VALUES
               ('snake', 'reptile'),
               ('frog', 'amphibian'),
               ('tuna', 'fish'),
               ('racoon', 'mammal')
            """)

print("{} rows were inserted".format(cursor.rowcount))
cursor.execute ("commit")
```

- Note that this code includes no error-checking, we will place it in a `try` block; errors will trigger exceptions that are caught and handled in the corresponding `except` block.

- Some queries produce a count indicating the number of rows inserted, deleted or updated. The count is available in the cursor's **`rowcount`** attribute.

VII.4.3 Execution of select statements

- The SQL SELECT statement can be used to retrieve information from one or more table.
- As with the preceding statements, SELECT queries are issued using `execute()`. However, unlike SQL statements such as DROP, CREATE TABLE, UPDATE, DELETE or INSERT, SELECT queries generate a result set (a set of rows and columns that match you selection criteria) that you must retrieve.
- `execute()` only issues the query, it does not return the result set.
- You can use the method **`fetchone()`** to get the rows one at a time. `fetchone()` returns the next row of the result set as a `tuple`, or the value `None` if no more rows are available.
- You can use the method **`fetchall()`** to get them all at once. `fetchall()` returns the entire result set all at once as a tuple of tuples, or as an empty tuple if the result set is empty.
- Here's how to use `fetchone()` for row-at-a-time retrieval:

```
cursor.execute ("SELECT name, category FROM animal")
while True:
    row = cursor.fetchone ()
    if row == None:
        break
    print("{}, {}".format(row[0], row[1]))
print("{} rows were returned".format(cursor.rowcount))
```

- In the above example, the loop test the value returned by `fetchone()`and exits when the result set has been exhausted.
  For each row returned, the tuple contains two values (that's how many columns the SELECT query asked for), which are then printed.
  After displaying the query result, the script also prints the number of rows returned (available as the value of the `rowcount` attribute).
- The following example retrieves the result of the SELECT statement thanks to the `fetchall()` method: the script iterates through the row set that `fetchall()` returns:

```
cursor.execute ("SELECT name, category FROM animal")
rows = cursor.fetchall ()
for row in rows:
    print("{}, {}".format(row[0], row[1]))
print("{} rows were returned".format(cursor.rowcount))
```

- This code prints the row count by accessing `rowcount`, just as for the `fetchone()` loop. Another way to determine the row count when you use `fetchall()` is by taking the length of the value that it returns.

VII.4.4 Parameterised statements

- The DB API supports a placeholder capability that allows you to bind data values to special markers within the query string passed to the `execute()` or `executemany()` methods.
- This provides an alternative to embedding the values directly into the query, the corresponding statement are reusable and execute more efficiently.
- Say, for example, that you need to fetch the rows of database table `animal` where the column *`name`* equals the current value of Python variable *`aName`*.
- Assuming the cursor instance is named *`c`*, you could perform this task by using Python's string formatting method as follows:

```
c.execute('SELECT * FROM animal WHERE name={}'.format(aName))
```

- However, this is not the recommended approach. This approach generates a different statement string for each value of *`aName`*, requiring such statements to be parsed and prepared anew each time.
- With parameter substitution, you pass to `execute()` a single statement string, with a placeholder instead of the parameter value. This lets `execute()` perform parsing and preparation just once, giving potentially better performance.
- A DBAPI-compliant module has an attribute `paramstyle` that identifies the style of markers to use as placeholders for parameters.
- If a module's `paramstyle` attribute is `'qmark'`(it means that question marks are used to represent the placeholder), you can express the above query as:

```
c.execute('SELECT * FROM animal WHERE name=?', [aName])
```

- If the value of the `paramstyle` attribute is `'format'` it indicates standard string formatting (in the C tradition), so you insert `%s` where you want to splice in parameters, for example.

```
c.execute('SELECT * FROM animal WHERE name=%s', [aName])
```

- If the value of the `paramstyle` attribute is `'pyformat'` it indicates extended format codes, as used with dictionary splicing, such as `%(foo)s`.

```
c.execute('SELECT   *   FROM   animal   WHERE   name=%(aName)s',
{'aName':aName})
```

- If the value of the `paramstyle` attribute is `'numeric'` it means fields of the form `:1` or `:2` (where the numbers are the numbers of the parameters).

```
c.execute('SELECT * FROM animal WHERE name=:1', [aName])
```

- If the value of the `paramstyle` attribute is `'named'` it means fields like `:`*`foobar`*, where *`foobar`* is a parameter name.

```
c.execute('SELECT    *    FROM    animal    WHERE    name=:aName',
{'aName':aName})
```

- When `paramstyle` does not imply named parameters, the second argument of method execute is a sequence. When parameters are named, the second argument of method execute is a dictionary.

- If there is the need to execute several parameterised statements with different sets of arguments, the **`executemany()`** method could be useful.

  `executemany(statement,*parameters)`

- This method executes a SQL statement on the database, once for each item of the given *`parameters`*. *`parameters`* is a sequence of sequences when the module's `paramstyle` is `'format'`, `'numeric'`, or `'qmark'`, and a sequence of dictionaries when `'named'` or `'pyformat'`.

- For example, the statement:

```
c.executemany('UPDATE atable SET x=? WHERE y=?',
                    (12,23),(23,34))
```

  that uses the `paramstyle` is `'qmark'` is equivalent to, but probably faster than, the two statements:

```
c.execute('UPDATE atable SET x=12 WHERE y=23')
c.execute('UPDATE atable SET x=23 WHERE y=34')
```

VII.4.5 Types

- Parameters passed to the database via placeholders must typically be of the right type. This means Python numbers (integers or floating-point values), strings (plain or Unicode), and `None` to represent SQL NULL.

- Python has no specific types to represent dates, times, and binary large objects (BLOBs). A DBAPI-compliant module supplies factory functions to build such objects.

- For example, if you want to add a date to a database, it should be constructed with (for example) the `Date` factory function of the corresponding database connectivity module.

- That allows the DB module to perform any necessary transformations behind the scenes.

- Each module is required to implement the factory functions and special values shown here:

`Date(year, month, day)`	Creates an object holding a date value
`Time(hour, minute, second)`	Creates an object holding a time value
`Timestamp(y,mon,d,h,min, s)`	Creates an object holding a timestamp value
`DateFromTicks(ticks)`	Creates an object holding a date value from ticks since epoch
`TimeFromTicks(ticks)`	Creates an object holding a time value from ticks
`TimestampFromTicks(ticks)`	Creates an object holding a timestamp value from ticks
`Binary(string)`	Creates an object holding a binary string value
`STRING`	Describes string-based column types (such as CHAR)
`BINARY`	Describes binary columns (such as LONG or RAW)
`NUMBER`	Describes numeric columns
`DATETIME`	Describes date/time columns
`ROWID`	Describes row ID columns

VIII TKINTER GUIS

- A lot of applications interact with users through a graphical user interface (GUI).

- A GUI is normally programmed through a toolkit, which is a library that implements **controls** (also known as ***widgets***) that are visible objects such as *buttons*, *labels*, *text entry* fields, and *menus*.

- A GUI toolkit lets you compose controls into a coherent whole, display them on-screen, and interact with the user, receiving input via such devices as the keyboard and mouse.

- Python gives you a choice among dozens of GUI toolkits. Some are platform-specific, but most are cross-platform to different degrees.

- The most widespread Python GUI toolkit is **`tkinter`**.

- `tkinter` is an object-oriented Python wrapper around the cross-platform toolkit **`Tk`**, which is also used with other scripting languages such as `Tcl` (for which it was originally developed) and `Perl`.

VIII.1 tkinter Fundamentals

- The `tkinter` module makes it easy to build simple GUI applications.

- You simply need to:

    1. import `tkinter`,
    2. create, configure, and position the widgets you want,
    3. enter the `tkinter` ***main loop***.

- Your application becomes event-driven, which means that the user interacts with the widgets, causing events, and your application responds via the functions you installed as handlers for these events.

- The following example shows a simple application that exhibits this general structure:

```
import sys, tkinter
tkinter.Label(text="Welcome!").pack()
tkinter.Button(text="Exit", command=sys.exit).pack()
tkinter.mainloop()
```

- The calls to `Label` and `Button` create the respective widgets and return them as results. Since we specify no parent windows, `tkinter` puts the widgets directly in the application's main window.

- The named arguments specify each widget's configuration. In this simple case, we don't need to bind variables to the widgets. We just call the **`pack()`** method on each widget, handing control of the widget's geometry to a ***layout manager*** object known as the **packer**.

- A ***layout manager*** (or ***geometry manager***) is an invisible component whose job is to position and to size widgets within other widgets (known as container or parent widgets), handling geometrical layout issues.
  The previous example passes no arguments to control the packer's operation, so therefore the packer operates in a default way.

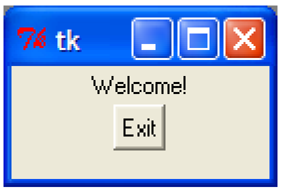
- When the user clicks on the button, the *`command`* callable of the `Button` widget executes without arguments. The example passes function `sys.exit` as the argument named command when it creates the `Button`. Therefore, when the user clicks on the button, `sys.exit()` executes and terminates the application.

- After creating and packing the widgets, the example calls `tkinter`'s **`mainloop()`** function, and thus enters the `tkinter` main loop and becomes event-driven.

- Since the only event for which the example installs a handler is a click on the button, nothing happens from the application's viewpoint until the user clicks the button.

- Meanwhile, however, the `tkinter` toolkit responds in the expected way to other user actions, such as moving the `tkinter` window, covering and uncovering the window, and so on.

- When the user resizes the window, the packer layout manager works to update the widgets' geometry. In this example, the widgets remain centered, close to the upper edge of the window, with the label above the button.



Note: this chapter refers to several all-uppercase, multi-letter identifiers (e.g., LEFT, RAISED, ACTIVE). All these identifiers are constant attributes of module `tkinter`, used for a wide variety of purposes.
If your code uses **`from tkinter import *`**, you can then use the identifiers directly. If not, you need to qualify those identifiers by preceding them with `tkinter.`.

VIII.2 Widget Fundamentals

- The `tkinter` module supplies many kinds of widgets, and most of them have several things in common.

- All widgets are instances of classes that inherit from class **`Widget`**. Class `Widget` itself is abstract; that is, you never instantiate `Widget` itself. You only instantiate concrete subclasses corresponding to specific kinds of widgets.

- To instantiate any kind of widget, you use the widget's constructor. The first argument of the constructor is always the parent window of the widget (the widget's master). If you omit this positional argument, the widget's master is the application's main window.

- All other arguments are in named form, *`option=value`*.

- After having created a widget *w*, you can set or change its options by calling ***`w`*`.config(`*`option=value`*`)`**.

- You can get an option of `w` by calling **`w.cget('`*`option`*`')`**, which returns the option's value.

- Each widget `w` is a mapping, so you can also get an option as `w['option']` and set or change it with `w['option']=value`.

VIII.2.1 Common Widget Options

- Many widgets accept some common options.

VIII.2.1.1 Color options

- `tkinter` represents colors with strings. The string can be a color name, such as 'red' or 'orange', or it may be of the form '#RRGGBB', where each of R, G, and B is a hexadecimal digit, to represent a color by the values of red, green, and blue components on a scale of 0 to 255

- Here are examples of color options:

`activebackground`

Background color for the widget when the widget is active, meaning that the mouse is over the widget and clicking on it makes something happen

`activeforeground`

Foreground color for the widget when the widget is active

`background` (also **`bg`**)

Background color for the widget

`foreground` (also `fg`)

Foreground color for the widget

VIII.2.1.2 Length options

- `tkinter` normally expresses a length as an integer number of pixels.

- Examples of common length options are:

`borderwidth`

Width of the border (if any), giving a 3D look to the widget

`padx`, `pady`

Extra space the widget requests from its geometry manager beyond the minimum the widget needs to display its contents, in the x and y directions

VIII.2.1.3 Options expressing numbers of characters

- Some options indicate a widget's requested geometry not in pixels, but rather as a number of characters, using average width or height of the widget's fonts:

`height`

Desired height of the widget; must be greater than or equal to 1

`width`

Desired width of the widget (when less than or equal to 0, desired width is just enough to hold the widget's current contents)

VIII.2.1.4 Other common options

- Other options accepted by many kinds of widgets are a mixed bag, dealing with both behavior and presentation issues.

`anchor`

Where the information in the widget is displayed; must be N, NE, E, SE, S, SW, W, NW, or CENTER (all except CENTER are compass directions)

`command`

Callable without arguments; executes when the user clicks on the widget (only for widgets `Button`, `Checkbutton`, and `Radiobutton`)

`font`

Font for the text in this widget

`image`

An image to display in the widget instead of text; the value must be a `tkinter` image object

`text`

The text string displayed by the widget

`textvariable`

The `tkinter` variable object associated with the widget

VIII.2.2 Common Widget Methods

- A widget `w` supplies many methods. Besides event-related methods, mentioned later in this chapter, commonly used widget methods are the following.

`w.cget(option)`

Returns the value configured in w for option.

`w.config(**options)`

`w.config()`, without arguments, returns a dictionary where each possible option of `w` is mapped to a tuple that describes it. Called with one or more named arguments, `config()` sets those options in `w`'s configuration.

`w.mainloop()`

Enters a `tkinter` event loop. Event loops may be nested; each call to mainloop enters one further-nested level of the event loop.

`w.quit()`

Quits a `tkinter` event loop. When event loops are nested; each call to quit exits one nested level of the event loop.

`w.update()`

Handles all pending events. Never call this while handling an event!

`w.wait_variable(v)`

`v` must be a `tkinter` variable object (covered in the next section). `wait_variable()` returns only when the value of `v` changes. Meanwhile, other parts of the application remain active.

`w.winfo_height()`

Returns `w`'s height in pixels.

`w.winfo_width()`

Returns `w`'s width in pixels.

VIII.2.3 tkinter Variable Objects

- The `tkinter` module supplies classes whose instances represent variables. Each class deals with a specific data type: **`DoubleVar`** for `float`, **`IntVar`** for `int`, **`StringVar`** for `str`.
- You can instantiate any of these classes without arguments to obtain an instance `x`, also known in `tkinter` as a variable object.
- Then, **`x.set(`*`datum`*`)`** sets `x`'s value to the given value, and **`x.get()`** returns `x`'s current value.
- You can pass `x` as the **`textvariable`** or **`variable`** configuration option for a widget. Once you do this, the widget's text changes to track any change to `x`'s value, and `x`'s value, in turn, tracks changes to the widget (for some kinds of widgets).
- A single `tkinter` variable can control more than one widget.

VIII.2.4 tkinter Images

- The `tkinter` class **`PhotoImage`** supports Graphical Interchange Format (GIF) and Portable PixMap (PPM) images.
- You instantiate class `PhotoImage` with a named argument **`file`**`=path` to load the image's data from the image file at the given `path` and get an instance `x`.
- You can then set `x` as the **`image`** configuration option for one or more widgets (a `Label` for instance) . When you do this, the widget displays the image rather than text.
- Being set as the image configuration option of a widget does not suffice to keep instances of `PhotoImage` alive. Be sure to hold such instances in a Python container object, typically a list or dictionary, to ensure that the instances are not garbage-collected.

VIII.3 Commonly Used Simple Widgets

- The `tkinter` module provides a number of simple widgets that cover most needs of basic GUI applications. This section documents the **`Button`**, **`Checkbutton`**, **`Entry`**, **Label**, **`Listbox`**, **`Radiobutton`**, **`Scale`**, and **`Scrollbar`** widgets.

VIII.3.1 Button

- Class **`Button`** implements a pushbutton, which the user clicks to execute an action. You instantiate `Button` with option **`text`**`=somestring` to let the button show text, or **`image`**`=imageobject` to let the button show an image.

- You normally use option **`command`**`=callable` to have *`callable`* execute without arguments when the user clicks the button.

- Besides methods common to all widgets, an instance `b` of class `Button` supplies the method.

  `b.invoke()`

  Calls without arguments the callable object that is `b`'s `command` option (emulates a button click).

VIII.3.2 Checkbutton

- Class **`Checkbutton`** implements a checkbox. You normally instantiate `Checkbutton` with exactly one of the two options **`text`** or **`image`**, to label the checkbox with a text or with an image.

- Optionally, use option **`command`**`=`*`callable`* to have *`callable`* execute without arguments when the user clicks the box.

- An instance `c` of `Checkbutton` must be associated with a `tkinter` variable object `v`, using configuration option **`variable`**`=v` of `c`. Normally, `v` is an instance of `IntVar`, and `v`'s value is 0 when the box is unchecked, and 1 when the box is checked.

- Besides methods common to all widgets, an instance `c` of class `Checkbutton` supplies five checkbox-specific methods.

`c.deselect()`
`c.select()`
    To programmatically select or deselect the checkbox

`c.invoke()` (see Button)

`c.toggle()`
    Toggles the state of `c`'s.

VIII.3.3 Entry

- Class **`Entry`** implements a text entry field (i.e., a widget in which the user can input and edit a line of text).

- An instance `e` of `Entry` supplies several methods and configuration options allowing fine-grained control of widget operation and contents, but in most cases you can get by with just three `Entry`-specific methods:

```
e.delete(0, END)              # clear the widget's contents
e.insert(END, somestring)     # append somestring to the widget's contents
somestring = e.get()          # get the widget's contents
```

Note: To display more than one line of text, use the `Text` widget.

VIII.3.4 Label

- Class **`Label`** implements a widget that just displays text or an image without interacting with user input. You can instantiate `Label` either with option **`text`** to let the widget display text, or **`image`** to let the widget display an image.

VIII.3.5 Listbox

- Class **`Listbox`** displays textual items and lets the user select one or more items. To set the text items for an instance `L` of class `Listbox`, in most programs you can get by with just two `Listbox`-specific methods:

```
L.delete(0, END)              # clear the listbox's items
L.insert(END, somestring)     # add somestring to the listbox's items
```

- Option **`selectmode`** defines the selection mode of a `Listbox`: for instance, SINGLE selection list or MULTIPLE selection list.

- An instance `L` of class `Listbox` supplies three selection-related methods.

`L.curselection()`

Returns a sequence of zero or more indices, from 0 upwards, of selected items.

`L.select_clear(i,j=None)`

Deselects the `i` item (all items from the `i` to the `j`).

`L.select_set(i,j=None)`

Selects the `i` item (all items from the `i` to the `j`).

VIII.3.5 Radiobutton

- Class `Radiobutton` implements a little box that is optionally checked. The user clicks the radiobutton to toggle it on or off. Radiobuttons come in groups: checking a radiobutton automatically unchecks all other radiobuttons of the same group.

- The `text` and `image` and `command` options behave as expected.

- An instance of `Radiobutton` must be associated with a `tkinter` variable, using the configuration option `variable` (see VIII.3.2).

- Note that `Radiobutton` instances form a group if, and only if, they share the same value for the variable option.

- The class `Radiobutton` method we have already described: `deselect()`, `invoke()`, `select()`.

VIII.3.6 Scale

- The class **`Scale`** implements a widget in which the user can input a value by sliding a cursor along a line.

- `Scale` supports configuration options to control the widget's looks and the value's range, but in most programs the only option you specify is **`orient`**`=HORIZONTAL` when you want the line to be horizontal (by default, the line is vertical).

- Besides methods common to all widgets, an instance `s` of class `Scale` supplies two scale-specific methods.

`s.get()`

Returns the current position of s's cursor, normally on a scale of 0 to 100.

`s.set(p)`

Sets the current position of s's cursor, normally on a scale of 0 to 100.

VIII.3.7 Scrollbar

- The class **`Scrollbar`** implements a widget similar to class `Scale`, almost always used to scroll another widget (most often a `Listbox`, or a `Text` or `Canvas`) rather than to let the user input a value.

- A `Scrollbar` instance `s` is connected to the widget that `s` controls (e.g., a `Listbox` instance `L`) through one configuration option on each of `s` and `L`.

- Exactly for this purpose, the widgets most often associated with a scrollbar supply a method named **`yview()`** and a configuration option named `yscrollcommand` for vertical scrolling (for horizontal scrolling, a method named **`xview()`** and a configuration option named `xscrollcommand`).

- For vertical scrolling, use `s`'s option **`command`**`=L.yview` so that user actions on `s` call `L`'s bound method `yview` to control `L`'s scrolling, and also use `L`'s option **`yscrollcommand`**`=s.set` so that changes to `L`'s scrolling, in turn, adjust the way `s` displays by calling `s`'s bound method `set()`.

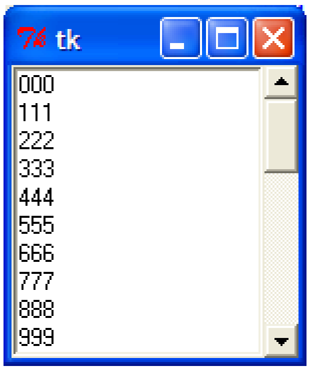
- The following example uses a `Scrollbar` to control vertical scrolling of a `Listbox`:

```
import tkinter

s = tkinter.Scrollbar()
L = tkinter.Listbox()
s.pack(side=tkinter.RIGHT, fill=tkinter.Y)
L.pack(side=tkinter.LEFT, fill=tkinter.Y)
s.config(command=L.yview)
L.config(yscrollcommand=s.set)

for i in range(30): L.insert(tkinter.END, str(i)*3)

tkinter.mainloop()
```



```
000
111
222
333
444
555
666
777
888
999
```

VIII.3.8 The Canvas Widget

- Class **`Canvas`** is a powerful, flexible widget used for many purposes, including plotting and, in particular, building custom widgets.

- Coordinates within a `Canvas` instance `c` are in pixels, with the origin at the upper left corner of `c` and positive coordinates growing rightward and downward.

- What you draw on a Canvas instance `c` are canvas items, which can be lines, polygons, `tkinter` images, arcs, ovals, texts, and others.

- Each item has an *item handle* by which you can refer to the item.

- You create a canvas item by calling on `c` a method with a name of the form **`create_`*`kindofitem`*()**, which returns the new item's handle.

VIII.4 Container Widgets

- The `tkinter` module supplies widgets whose purpose is to contain other widgets.
- A **`Frame`** instance does nothing more than act as a container.
- A **`Toplevel`** instance is a top-level window, so your window manager interacts with it.
- To ensure that a widget parent, which must be a `Frame` or `Toplevel` instance, is the parent of another widget child, pass parent as the first parameter when you instantiate child.

VIII.4.1 Frame

- Class **`Frame`** represents a rectangular area of the screen contained in other frames or top-level windows.
- `Frame`'s only purpose is to contain other widgets. Option `borderwidth` defaults to 0, so an instance of `Frame` normally displays no border.

VIII.4.2 Toplevel

- Class **`Toplevel`** represents a rectangular area of the screen that is a top-level window and therefore receives decoration from whatever window manager handles your screen.
- Each instance of `Toplevel` can interact with the window manager and can contain other widgets.
- Every program using `tkinter` has at least one top-level window, known as the **root window**.
- You can instantiate `tkinter`'s root window explicitly using **`root=tkinter.Tk()`**; otherwise `tkinter` instantiates its root window implicitly as and when first needed.
- If you want to have more than one top-level window, first instantiate the main one with `root=tkinter.Tk()`. Later in your program, you can instantiate other top-level windows as needed, with calls such as `another_toplevel=tkinter.`**`Toplevel()`**`.`
- An instance `T` of class `Toplevel` supplies many methods enabling interaction with the window manager. The cross-platform methods used most often are as follows.

`T.deiconify()`

Makes `T` display normally, even if previously `T` was iconified or invisible.

`T.geometry([geometry_string])`

Without arguments, returns a string encoding `T`'s size and position: *`widthxheight+x_offset+y_offset`*. `T.geometry(S)`, with one argument `S` (a string of the same form), sets `T`'s size and position according to `S`.

`T.iconify()`

Makes T display as an icon.

`T.maxsize([width,height])`
`T.minsize([width,height])`

Set or get the maximum (minimum) size of `T`.

`T.resizable([width,height])`

Without arguments, returns a pair of integers (each 0 or 1) whose two items indicate if user action via the window manager can change `T`'s width and height, respectively. `T.resizable(W,H)`, with two integer arguments `W` and `H` (each 0 or 1), sets the user's ability to change T's width and height according to the truth values of W and H.

`T.title([title_string])`

Set or get the title of the window.

`T.withdraw()`

Makes T invisible.

VIII.5 Menus

- Class **`Menu`** implements all kinds of menus: menubars of top-level windows, submenus, and pop-up menus.

- To use a `Menu` instance `m` as the menubar for a top-level window `w`, set `w`'s configuration option **`menu`**=`m`. To use `m` as a submenu of a `Menu` instance `x`, call `x`**`.add_cascade()`** with a named argument `menu=m`. To use `m` as a pop-up menu, call method **`m.post()`**.

- Besides configuration options covered earlier in this chapter, a `Menu` instance `m` supports option **`postcommand`**=`callable`. `tkinter` calls *`callable`* without arguments each time it is about to display `m`.

VIII.5.1 Menu-Specific Methods

- Besides methods common to all widgets, an instance `m` of class `Menu` supplies several menu-specific methods.

`m.add(entry_kind, **entry_options)`

Adds after `m`'s existing entries a new entry whose kind is the string *`entry_kind`*, which is one of the strings 'cascade', 'checkbutton', 'command', 'radiobutton', or 'separator'.

`m.delete(i[,j])`

Removes `m`'s `i` entry. `m.delete(i,j)` removes `m`'s entries from the `i` one to the `j` one, included. The first entry has index 0.

`m.invoke(i)`

Invokes m's `i` entry, just as if the user clicked on it.

`m.post(x,y)`

Displays m as a pop-up menu, with `m`'s upper left corner at coordinates `x`,`y` (offsets in pixels from upper left corner of `tkinter`'s root window).

`m.unpost()`

Closes `m` if `m` was displaying as a pop-up menu, otherwise does nothing.

VIII.5.2 Menu Entries

- When a menu `m` displays, it shows a vertical (horizontal for a menubar) list of entries. Each entry can be one of the following kinds:

`cascade`

A submenu.

`checkbutton`

Similar to a `Checkbutton` widget

`command`

Similar to a Button widget

`radiobutton`

Similar to a `Radiobutton` widget

`separator`

A line segment that separates groups of other entries

- The following example shows how to add a *menubar* with typical File and Edit menus:

VIII.6 The Text Widget

- Class `Text` implements a multiline text editor, able to display images as well as text in one or more fonts and colors.
- An instance `t` of `Text` supports many ways to refer to specific points in `t`'s contents.
- In some very simple cases, you can get by with just three Text-specific idioms:

```
t.delete('1.0', END)                # clear the widget's contents
t.insert(END, astring)              # append astring to the widget's contents
somestring = t.get('1.0', END)      # get the widget's contents as a string
```

VIII.6.1 Text Widget Methods

- An instance t of class Text supplies many methods.

`t.delete(i[,j])`

Removes `t`'s character at index `i` or from `i` to index `j`, included.

`t.get(i[,j])`

Returns t's character at index `i` or from `i` to index `j`, included.

`t.image_create(i,**window_options)`

Inserts an embedded image in `t`'s contents at index `i`.

`t.insert(i,s)`

Inserts string `s` in `t`'s contents at index `i`.

`t.search(pattern,i,**search_options)`

Finds the first occurrence of string pattern in t's contents not earlier than index i and returns a string that is the index of the occurrence, or an empty string '' if not found.

VIII.6.2 Giving Text a Scrollbar

- You'll often want to couple a Scrollbar instance to a `Text` instance in order to let the user scroll through the text.
- The mechanism you are going to use to do so is similar to the one used in the `ListBox` example (see VIII.3.7)

VIII.6.3 Fonts

- You can change fonts on any `tkinter` widget with option **`font`**=*`font`*. In most cases it makes no sense to change widgets' fonts.

- Module **`tkFont`** supplies class `Font`, attributes `BOLD`, `ITALIC`, and `NORMAL` to define font characteristics, and functions families (returns a sequence of strings naming all families of available fonts) and names (returns a sequence of strings naming all user-defined fonts).

- Frequently used font options are:

`family`	Font family (e.g. 'courier' or 'helvetica')
`size`	Font size (in points if positive, in pixels if negative)
`slant`	NORMAL (default) or ITALIC
`weight`	NORMAL (default) or BOLD

VIII.7 Geometry Management

- In all the examples so far, we have made each widget visible by calling method **`pack()`** on the widget. This is representative of real-life `tkinter` usage. However, two other layout managers exist and are sometimes useful.

Note: Never mix geometry managers for the same container widget: all children of each given container widget must be handled by the same geometry manager, or strange effects may result.

VIII.7.1 The Packer

- Calling method **`pack()`** on a widget delegates widget geometry management to a simple and flexible layout manager component called the **`Packer`**.

- The `Packer` sizes and positions each widget within a container (parent) widget, according to each widget's space needs (including options `padx` and `pady`).

- Each widget `w` supplies the following Packer-related methods.

`w.pack(**pack_options)`

Delegates geometry management to the packer. `pack_options` may include:

`expand`
When true, `w` expands to fill any space not otherwise used in w's parent.

`fill`
Determines whether `w` fills any extra space allocated to it by the packer, or keeps its own minimal dimensions: `NONE` (default), `X` (fill only horizontally), `Y` (fill only vertically), or `BOTH` (fill both horizontally and vertically).

`side`
Determines which side of the parent `w` packs against: `TOP` (default), `BOTTOM`, `LEFT`, or `RIGHT`. When more than one child requests the same side, the rule is first come, first served.

`w.pack_forget()`

The packer forgets about `w`. `w` remains alive but invisible, and you may show w again later (by calling `w.pack()` again).

VIII.7.2 The Gridder

- Calling method **`grid()`** on a widget delegates widget geometry management to a specialized layout manager component called the **`Gridder`**. The `Gridder` sizes and positions each widget into cells of a table (grid) within a container (parent) widget.

- Each widget `w` supplies the following Gridder-related methods.

`w.grid(**grid_options)`

Delegates geometry management to the gridder. grid_options may include:

`column`
The column to put `w` in; default 0 (leftmost column).

`columnspan`
How many columns `w` occupies; default 1.

`ipadx`, **`ipady`**
How many pixels to pad `w`, horizontally and vertically, inside w's borders.

`padx`, **`pady`**
How many pixels to pad `w`, horizontally and vertically, outside w's borders.

`row`
The row to put `w` in; default the first row that is still empty.

`rowspan`
How many rows `w` occupies; default 1.

`sticky`
What to do if the cell is larger than `w`. By default `w` is centered in its cell. `sticky` may be the string concatenation of zero or more of N, E, S, W, NE, NW, SE, and SW, compass directions indicating the sides and corners of the cell to which `w` sticks.

`w.grid_forget()`

The gridder forgets about `w`. `w` remains alive but invisible, and you may show `w` again later (by calling `w.grid()` again).

VIII.7.3 The Placer

- Calling method **`place()`** on a widget explicitly handles widget geometry management, thanks to a simple layout manager component called the **`Placer`**.

- The `Placer` sizes and positions each widget `w` within a container (parent) widget exactly as `w` explicitly requires. Other layout managers are usually preferable, but the `Placer` can help you implement custom layout managers. Each widget `w` supplies the following Placer-related methods.

`w.place(**place_options)`

Delegates geometry management to the placer. `place_options` may include:

`anchor`
The exact spot of w other options refer to: may be N, E, S, W, NE, NW, SE, or SW, compass directions indicating the corners and sides of w; default is NW (the upper left corner of w)

`bordermode`
INSIDE (the default) to indicate that other options refer to the parent's inside (ignoring the parent's border); OUTSIDE otherwise

`height`, **`width`**
Height and width in pixels

`relheight`, **`relwidth`**
Height and width as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget

`relx`, **`rely`**
Horizontal and vertical offset as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget

`x`, **`y`**
Horizontal and vertical offset in pixels

`w.place_forget()`

As expected ..

VIII.8 tkinter Events

- So far, we've seen only the most elementary kind of event handling: the callbacks performed on callables installed with the **`command=`** option of buttons and menu entries of various kinds.

- `tkinter` also lets you install callables to call back when needed to handle a variety of events. However, `tkinter` does not let you create your own custom events; you are limited to working with events predefined by `tkinter` itself.

VIII.8.1 The Event Object

- General event callbacks must accept one argument event that is a `tkinter` **`event`** object. Such an event object has several attributes describing the event:

  `char`
  A single-character string that is the key's code (only for keyboard events)

  `keysym`
  A string that is the key's symbolic name (only for keyboard events)

  `num`
  Button number (only for mouse-button events); 1 and up

  `x`, **`y`**
  Mouse position, in pixels, relative to the upper left corner of the widget

  `x_root` , **`y_root`**
  Mouse position, in pixels, relative to the upper left corner of the screen

  `widget`
  The widget in which the event has occurred

VIII.8.2 Binding Callbacks to Events

- To bind a callback to an event in a widget `w`, call **`w.bind()`**, describing the event with a string, usually enclosed in angle brackets ('<...>').

- The following example prints 'Hello World' each time the user presses the Enter key:

```
from tkinter import *

root = Tk()
def greet(*ignore): print('Hello World')
root.bind('<Return>', greet)
root.mainloop()
```

VIII.8.3 Event Names

- Frequently used event names, which are almost all enclosed in angle brackets, fall into a few categories.

VIII.8.3.1 Keyboard events

Key

> The user clicked any key. The event object's attribute `char` tells you which key, but for normal keys only, not for special keys. The event object's attribute `keysym` is equal to attribute char for letters and digits, is the character's name for punctuation characters, and is the key name for special keys, as covered in the next paragraph.

Special keys

> Special keys are associated with event names: F1, F2, ..., up to F12 for function keys; Left, Right, Up, Down for arrow keys; Prior, Next for page-up, page-down; BackSpace, Delete, End, Home, Insert, Print, Tab, for keys so labeled; Escape for the key often labeled Esc; Return for the key often labeled Enter; Caps_Lock, Num_Lock, Scroll_Lock for locking-request keys; Alt_L, Control_L, Shift_L for the modifier keys Alt, Ctrl, Shift (without distinction among the multiple instances of such modifier keys in a typical keyboard). All of these event names are placed within angle brackets.

Normal keys

> Normal keys are associated with event names without surrounding angle brackets—the only event names to lack such brackets. The event name of each normal key is just the associated character, such as 'w', '1', or '+'. Two exceptions are the Space key, whose event name is '<space>', and the key associated with the less-than character, whose event name is '<less>'.

- All key event names can be modified by prefixing 'Alt-', 'Shift-', or 'Control-'. In this case, the whole event name does always have to be surrounded with '<...>'. For example, '<Control-Q>' and '<Alt-Up>' name events corresponding to normal or special keys with modifiers.

VIII.8.3.2 Mouse events

Button-1, Button-2, Button-3

> The user pressed the left, middle, or right mouse-button. A two-button mouse produces only events Button-1 and Button-3, since it has no middle button.

B1-Motion, B2-Motion, B3-Motion

> The user moved the mouse while pressing the left, middle, or right mouse button (there is no mouse event for mouse motion without pressing a button, except for Enter and Leave).

ButtonRelease-1, ButtonRelease-2, ButtonRelease-3

The user released the left, middle, or right mouse button.

Double-Button-1, Double-Button-2, Double-Button-3

The user double-clicked the left, middle, or right mouse button (such an action also generates Button-1, Button-2, or Button-3 before the double-click event).

Enter

The user moved the mouse so that the mouse entered the widget.

Leave

The user moved the mouse so that the mouse exited the widget.

VIII.8.4 Event-Related Methods

- Each widget w supplies the following event-related methods.

`w.bind(event_name,callable[,'+'])`

Sets callable as the callback for *`event_name`* on `w`. *`w.bind(event_name,callable,'+')`* adds callable to the previous bindings for *`event_name`* on *`w`*.

`w.unbind(event_name)`

Removes all callbacks for `event_name` on `w`.

VIII.8.5 Other Callback-Related Methods

- Each widget w supplies the following other callback-related methods.

`w.after(ms,callable,*args)`

Starts a timer that calls `callable(*args)` about `ms` milliseconds from now. Returns an ID that you can pass to `after_cancel()` to cancel the timer.

`w.after_cancel(id)`

Cancels the timer identified by *`id`*.

IX SOCKET PROGRAMMING IN PYTHON

IX.1 Overview

- Python program can communicate over a network using different client-server protocols, the main ones are:

    - File Transfer Protocol (FTP) is supported thanks to the **`ftplib`** module.
    - Post Office Protocol - Version 3 (POP3) is supported thanks to the **`pop`** module.
    - Internet Message Access Protocol (IMAP) is supported thanks to the **`imaplib`** module.
    - Simple Mail Transfer Protocol (SMTP) is supported thanks to the **`smtplib`** module.
    - Network News Transfer Protocol (NNTP) is supported thanks to the **`nntplib`** module.
    - Telnet is supported thanks to the **`telnetlib`** module.
    - Simple Object Access Protocol (SOAP) is supported thanks to the **`SOAPpy`** module.

- Python's **`socket`** module is the foundational module for all network programming in Python. This chapter introduces how to use raw sockets to create network oriented applications in Python.

IX.2 Sockets programming

- According to the formal definition a socket is “An endpoint of communication to which a name may be bound”.

- Different kinds of sockets (called *domains* or *family*) do exist. The two main ones are:

  Unix Domain (AF_UNIX or Address Family UNIX)

  The sockets under this domain are used when two or more processes within a system have to communicate with each other.

  Internet Domain (AF_INET or Address Family Internet)

  This domain represents the processes that communicate over the IP protocol.
  The sockets created for this domain are represented using a (host, port) tuple.

- Regardless of which domain you are using, there are two different styles of socket connections.

  1. The first type is connection-oriented: a connection must be established before communication can occur. This type of communication is also referred to as a "***stream socket***."
     Connection-oriented communication offers sequenced, reliable, and unduplicated delivery of data, and without record boundaries.
     The primary protocol that implements such connection types is TCP. To create TCP sockets, one must use **`SOCK_STREAM`** as the type of socket one wants to create.

  2. The second type is the datagram type of socket, which is connectionless: no connection is necessary before communication can begin.
     Here, there are no guarantees of sequencing, reliability, or non-duplication in the process of data delivery. This lack of guarantees makes datagram socket usually provide better performance.
     The primary protocol that implements such connection types is UDP. To create UDP sockets, one must use **`SOCK_DGRAM`** as the type of socket one wants to create.

IX.3 How to create a TCP/IP Socket application

- The following are the steps necessary for creating an application that uses TCP/IP sockets:

  1. Importing the `socket` module
  2. Creating a socket
  3. Connecting the socket
  4. Binding the socket to an address
  5. Listening and accepting connections
  6. Transferring data/receiving data
  7. Closing the socket

IX.3.1 Creating a socket (client and server side)

- A socket can be created by making call to the **`socket(`***`family, type`***`)`** function.

- The `socket()` function returns a socket in the domain and type specified.

- The main parameters of the function are:

  `family`: the valid values are AF_UNIX for the UNIX domain and AF_INET for the Internet domain.
  `type`: the type of the protocol to be used: SOCK_STREAM for TCP, SOCK_DGRAM for UDP.

```
from socket import *
…
testsocket=socket(AF_INET,SOCK_STREAM)
```

- Sockets thus created can be used on the server-side or client-side.

IX.3.2 Binding the socket to an address (server side)

- If the socket has to be used on the server side, then the socket has to be bound to an address and a port, thus naming it.

- To bind a socket to an address, the **`bind()`** method of the socket object has to be used.

- The valid parameter is a tuple containing the address to which the socket has to be bound and the port at which it has to listen for incoming requests.

- Valid port numbers range from 0-65535, although those less than 1024 are reserved for the system. A list of well-known port numbers is accessible at this Web site: http://www.iana.org/assignments/port-numbers

```
testsocket.bind((’192.168.51.1’,8080))
```

Note: instead of an IP address or a machine name, you can use the value returned by the **`gethostname()`** method to retrieve the name of the server.

IX.3.3 Listening and accepting connections (server side)

- Once a socket has been named, it then must be instructed to listen at the given port for incoming requests.
- This can be done using the **`listen()`** method.
- `listen()` accepts as argument a number representing the maximum queued connection. The argument should be at least 1.

```
testsocket.listen(2)
```

- The next thing to be done is to accept the incoming connection requests using the **`accept()`** method.
- This method returns a tuple containing a new socket object representing the client and the address of the client.

```
clientsock, address= testsocket.accept()
```

- It is important to understand that the "server" socket doesn't send or receive any data; it just produces "client" sockets.
- Each client socket is created in response to some other "client" socket doing a `connect()` to the host and port the server socket is bound to.
- By default, `accept()` is blocking, meaning that execution is suspended until a connection arrives.
- Sockets do support a non-blocking mode (see IX.3.5)

IX.3.4 Connecting the socket (client side)

- To use the socket on the client side, it needs to be connected to a host using the **`connect()`** method of the socket object.
- The `connect()` method accepts either the host name as the parameter (for the AF_UNIX domain) or a tuple containing host name/address and port number as parameter (for the AF_INET domain).

```
testsocket.connect((’192.168.51.1’,8080))
```

Note: The socket class also provides the `connect_ex()` method. This method behaves like `connect()`, but returns an error indicator instead of raising an exception when errors are encountered during the connection.

IX.3.5 Transferring data/receiving data (client and server side)

- Data can be transferred using the **`recv()`** and **`send()`** methods of the socket object.

- The `recv()` method is used to receive the data sent from the server or from the client. The main parameter of this method is a buffer size, it represents the maximum amount of data to be received at once. The return value is a bytes object representing the data received.

```
buff=1024
testsocket.recv(buff)
```

- The **`recv_into(`*`buffer [,nbytes]`*`)`** method can also be used to receive the data.

- This method receives up to *`nbytes`* bytes from the socket, storing the data into a buffer rather than creating a new byes object. If *`nbytes`* is not specified (or 0), the method receives up to the size available in the given *`buffer`*.

- The **`send()`** method is used to send the data from the server or from the client. The main parameter of `send()` is the data (a bytes object) to be sent. The return value is the number of bytes sent.

```
data=input('>>')
testsocket.send(data.encode())
```

- When calling the `recv()` method, the script blocks until there is some data to read or until the connection is closed (and there is no timeout by default !).

- Sockets do support a non-blocking mode; the **`setblocking(`*`flag`*`)`** method can be used to control that. If flag is 0, the socket is set to non-blocking, else to blocking mode the default.

- In non-blocking mode, if a `recv()` call doesn't find any data, or if a `send()` call can't immediately dispose of the data, an exception **`socket.error`** is raised; in blocking mode, the calls block until they can proceed.

- Another possibility is to associate a timeout with the socket operations. The **`settimeout(`*`value`*`)`** method set a timeout on blocking socket operations. The value argument can be a positive float expressing seconds, or `None`. If a float is given, subsequent socket operations will raise a timeout exception if the timeout period value has elapsed before the operation has completed. Setting a timeout of `None` disables timeouts on socket operations.

- The method **`gettimeout()`** returns the timeout in floating seconds associated with socket operations, or `None` if no timeout is set.

IX.3.6 Closing the connection (client and server side)

- To end the communication, you must call the **`close()`** method of the socket object.

```
testsocket.close()
```

- Strictly speaking, you're supposed to use **`shutdown(`*`how`*`)`** on a socket before you close it. The shutdown is an advisory to the socket at the other end. Depending on the argument you pass it, it can mean "I'm not going to send anymore, but I'll still listen" (`SHUT_WR`), "I'm not listening, but I continue to send data" (`SHUT_RD`) or "I'm not going to send nor to listen anymore" (`SHUT_RDWR`).

- In Python, a `close()` is the same as `shutdown(); close()`. So in most situations, an explicit shutdown is not needed.

- Python takes the automatic shutdown a step further, and says that when a socket is garbage collected, it will automatically do a `close()` if it's needed. But relying on this is a very bad habit. If your socket just disappears without doing a `close()`, the socket at the other end may hang indefinitely, thinking you're just being slow.

IX.4 How to create a UDP/IP Socket application

- The following are the steps necessary for creating an application that uses TCP/IP sockets:

  1. Importing the `socket` module
  2. Creating a socket
  3. Connecting the socket
  4. Binding the socket to an address
  5. Waiting for a message to arrive
  6. Optionally sending a response to the message
  7. Closing the socket

IX.4.1 Creating a socket (client and server side)

- The socket is created by making call to the **`socket()`** function.

```
from socket import *
…
testsocket=socket(AF_INET,SOCK_DGRAM)
```

- Sockets thus created can be used on the server-side or client-side.

IX.4.2 Binding the socket to an address (server side)

- To bind a socket to an address and port, the **`bind()`** method of the socket object has to be used.

```
testsocket.bind((’192.168.51.1’,8080))
```

IX.4.3 Waiting for client messages (server side)

- The next thing to be done is to wait for the incoming messages (datagrams) using the **`recvfrom(`*`bufsize`*`)`** method.
  The maximum amount of data to be received at once is specified by *`bufsize`* (the value of *`bufsize`* should be a relatively small power of 2, for example, 4096).

- This method returns a tuple containing a pair (bytes, address) where bytes is a bytes object representing the data received and address is the address of the socket sending the data.

```
data, address= testsocket.recvfrom(2048)
```

- The **`recvfrom_into(`*`buffer`*`)`** method can also be used to retrieve the message. This method writes the message received into *buffer* instead of creating a new bytes object. The return value is a pair containing the number of bytes received and the address of the socket sending the data.

- If necessary, the server will then send a response to the client using the **`sendto()`** method.

IX.4.4 Sending messages to the server (client side)

- After having created a socket the client uses the method **`sendto(`*`bytes, address`*`)`** to send a message to the server. The destination socket is specified by *`address`*. This method returns the number of bytes sent.

```
testsocket.sendto(b’a message’, ’128.178.115.56’)
```

IX.4.5 Closing the connection (client and server side)

- To end the communication, you must call the **`close()`** method of the socket object.

```
testsocket.close()
```

IX.5 The socketserver Module

IX.5.1 Overview

- The Python library supplies a framework module, **`socketserver`**, to help you implement simple Internet servers.

- `socketserver` supplies server classes **`TCPServer`**, for connection-oriented servers using TCP, and **`UDPServer`**, for datagram-oriented servers using UDP, with the same interface.

- `TCPServer` and `UDPServer` supplies many attributes and methods, and you can subclass either class and override some methods to architect your own specialized server framework.

- Classes `TCPServer` and `UDPServer` implement synchronous servers that can serve one request at a time. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request

- Classes **`ThreadingTCPServer`** and **`ThreadingUDPServer`** implement threaded servers, spawning a new thread per request. You are responsible for synchronizing the resulting threads as needed.

- Classes **`ForkingTCPServer`** and **`ForkingUDPServer`** implement servers that fork (run) a new process, per request. In a forking server, a child process is forked off for every client connection. The parent process keeps listening for new connections, while the child deals with the client. When the client is satisfied, the child process simply exits. Because the forked processes run in parallel, the clients don’t have to wait for each other.
  Forking can be a bit resource-intensive (each forked process needs its own memory).

IX.5.2 An example

- For normal use of `socketserver`, you subclass the **`BaseRequestHandler`** class provided by `socketserver` and override the **`handle()`** method.

- Then you instantiate a server class (`TCPServer`, `ThreadingTCPServer`, …), passing the address pair on which to serve and your subclass of `BaseRequestHandler`.

- Finally, you call **`serve_forever()`** on the server instance.

- The class `BaseRequestHandler` supplies the following methods and attributes.

  `client_address`: the pair (host,port) associated with the client, set by the base class at connection.

  `handle()`: our subclass overrides this method, and the server calls the method on a new instance of your subclass for each incoming request. For a TCP server, your implementation of handle conducts a conversation with the client on socket `request` to service the request. For a UDP server, your implementation of handle examines the datagram in `request[0]` and sends a reply with `request[1].sendto()`.

  `request`: for a TCP server, the `request` attribute is the socket connected to the client. For a UDP server, the `request` attribute is a pair *`(data,sock)`*, where *`data`* is a bytes objects representing the data the client sent as a request (up to 8,192 bytes) and *`sock`* is the server socket. Your handle method can call method `sendto()` on *`sock`* to send a reply to the client.

  `server`: the `server` attribute is the server instance that instantiated this handler object.

- If you are working with a stream (which you probably are, if you use `TCPServer`), you can use the class **`StreamRequestHandler`**, which sets up two other attributes, **`rfile`** (for reading) and **`wfile`** (for writing). You can then use these file-like objects to communicate with the client.

- The following example creates a Threaded TCP echo server using the `socketserver` module:

```
import socketserver

class EchoHandler(socketserver.BaseRequestHandler):
    def handle(self):
        print("Connected from", self.client_address)
        while True:
            receivedData = self.request.recv(8192)
            if not receivedData: break
            self.request.sendall(receivedData)
        self.request.close( )
        print("Disconnected from", self.client_address)

srv = socketserver.ThreadingTCPServer(('',8085), EchoHandler)
srv.serve_forever()
```

- The following client program could be used to test the server:

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('localhost', 8085))
print("Connected to server")
data = """A few lines of data
to send to the
server"""
for line in data.splitlines():
    sock.sendall(line.encode())
    print("Sent:", line)
    response = sock.recv(8192)
    print("Received:", response.decode())
sock.close()
```

IX.5.3 Multiple Connections

- With the server classes `TCPServer` and `UDPServer` only one client could connect and get its request handled at a time.

- If one request may takes a bit of time, it's important that more than one connection can be dealt with simultaneously.

- There are three main ways of achieving this: forking, threading, and asynchronous I/O.

- Forking and threading can be dealt with very simply, by using mix-in classes with any of the `socketserver` servers (`ForkingTCPServer`, `ThreadingTCPServer`, …). Even if you want to implement them yourself, these methods are quite easy to work with. They do have their drawbacks, however:
    forking takes up resources, and may not scale well if you have very many clients,
    threading can lead to synchronization problems.

- Asynchronous I/O is a bit more difficult to implement at a low level. The basic mechanism is the **`select()`** and **`poll()`** functions (of the two, `poll()` is more scalable, but it is only available in UNIX systems) of the **`select`** module (which is quite hard to deal with).

- Luckily, frameworks exist that work with asynchronous I/O on a higher level, giving you a simple, abstract interface to a very powerful and scalable mechanism.
  A basic framework of this kind, which is in included in the standard library, consists of the **`asyncore`** and **`asynchat`** modules.

- Another option is to use the **`Twisted`** (non-standard) python module.
  `Twisted` is an asynchronous networking framework written in Python that supports TCP, UDP, multicast, SSL/TLS, serial communication, and more. It supports both clients and servers and includes implementations of a number of commonly used network services such as a web server, an IRC chat server, a mail server, a relational database interface, and an object broker.

- Although Twisted supports processes and threads for longer-running actions, it also uses an asynchronous, event-driven model to handle clients, which is similar to the event loop of GUI libraries like tkinter. In fact, it abstracts an event loop, which multiplexes among open socket connections coincidentally.

X RESTFUL WEB SERVICES

- **REST** stands for **Representational State Transfer**, which is an architectural style for networked hypermedia applications, it is primarily used to build Web services that are lightweight, maintainable, and scalable.
  A service based on REST is called a **RESTful service**.

- REST is not dependent on any protocol, but almost every RESTful service uses HTTP as its underlying protocol.

- An API (application programming interface) is a set of subroutine definitions, protocols, and tools for building application software. Web service APIs that adhere to the REST architectural constraints are called **RESTful APIs**

Note: there is no "official" standard for RESTful web APIs: REST is an architectural style, not a standard in itself, but RESTful implementations make use of standards, such as HTTP, URI, JSON, and XML.

X.1 Features of a RESTful Services

- Every system uses resources. These resources can be pictures, video files, Web pages, business information, or anything that can be represented in a computer-based system. The purpose of a service is to provide a window to its clients so that they can access these resources. Service architects and developers want this service to be easy to implement, maintainable, extensible, and scalable. A RESTful design promises that and more.

- In general, RESTful services should have the following properties and features:

    - Representations
    - Messages
    - URIs
    - Uniform interface
    - Stateless
    - Links between resources
    - Caching

X.1.1 Representations

- The focus of a RESTful service is on resources and how to provide access to these resources. You can use any format for representing the resources (JSON, XML, HTML, Text, …), as REST does not put a restriction on the format of a **representation**.

- For example a resource called "Point" can be represented like this using JSON:
  {"x": 12,"y": 23}
  or like that using XML:
  <point><x>12</x><y>23</y></point>

- In fact, a service may you use more than one format and decide which one to use for a response depending on the type of client or some request parameters.

X.1.2 Messages

- The client and service talk to each other via **messages**. Clients send a request to the server, and the server replies with a response. Apart from the actual data, these messages also contain some metadata about the message. RESTFul web services usually do use HTTP messages.

- An HTTP request message is composed of the following parts :
  VERB : one of the HTTP methods : GET, PUT, POST, DELETE, OPTIONS, etc
  URI : the URI of the resource on which the operation is going to be performed.
  HTTP Version : the version of HTTP
  Request Header : the metadata as a collection of key-value pairs like: client type, the formats client supports, format type of the message body, cache settings for the response, and a lot more information.
  Request Body : the actual message content : the payload. In a RESTful service, that's where the representations of resources sit in a message.

- An HTTP response is composed of the following parts :
  HTTP Version : the version of HTTP
  Response code: the status of the request (a 3-digit HTTP status code).
  Response Header : contains the metadata and settings about the response message.
  Response Body : contains the representation if the request was successful.

X.1.3 Addressing Resources

- REST requires each resource to have at least one URI.

- A RESTful service uses a directory hierarchy like, human readable URIs to address its resources.

- A URI do identify a resource or a collection of resources. The actual operation is determined by an HTTP verb. The URI should not say anything about the operation or action. This enables us to call the same URI with different HTTP verbs to perform different operations.

- Suppose we have a database of persons and we wish to expose it to the outer world through a service. A resource person can be addressed like this:

  ```
  http://MyService/Persons/1
  ```

- This URI has following format: Protocol://ServiceName/ResourceType/ResourceID

X.1.4 Query Parameters in URI

- The following URI is constructed with the help of a query parameter:

  `http://MyService/Persons?id=1`

- In this example, the query parameter approach works just fine and REST does not stop you from using query parameters. However, compare to the URI `http://MyService/Persons/1`, this approach has a few disadvantages: increased complexity, reduced readability, bad SEO support.

- You should use query parameters only for the use they are intended for: providing parameter values to a process. For example, if you want the format of the presentation to be decided by the client, you can achieve that through a parameter like this:

  `http://MyService/Persons/1?format=xml&encoding=UTF8`

or

  `http://MyService/Persons/1?format=json&encoding=UTF8`

- Including the parameters format and encoding here in the main URI in a parent-child hierarchy will not be logically correct as they have no such relation:

  `http://MyService/Persons/1/json/UTF8` (not optimal ...)

X.1.5 Uniform Interface

- RESTful systems should have a uniform interface. HTTP 1.1 provides a set of methods, called ***verbs***, for this purpose. Among these the more important verbs are:

  `GET` : **read** a resource.
  `PUT` : insert a new resource or **update** it if the resource already exists.
  `POST` :**insert** a new resource. Also can be used to update an existing resource.
  `DELETE` : **delete** a resource .
  `OPTIONS` : list the allowed operations on a resource.
  `HEAD` : return only the response headers and no response body.

- You should use these methods only for the purpose for which they are intended. For instance, never use GET to create or delete a resource on the server.

X.1.6 Statelessness

- A RESTful service is stateless and does not maintain the application state for any client. A request cannot be dependent on a past request and a service treats each request independently.

- Stateless services are easier to host, easy to maintain, and more scalable. Plus, such services can provide better response time to requests, as it is much easier to load balance them.

X.1.7 Links between resources

- A resource representation can contain links to other resources like an HTML page contains links to other pages. The representations returned by the service should drive the process flow as in case of a website.

X.1.8 Caching

- Caching is the concept of storing the generated results and using the stored results instead of generating them repeatedly if the same request arrives in the near future. This can be done on the client, the server, or on any other component between them, such as a proxy server. Caching is a great way of enhancing the service performance, but if not managed properly, it can result in client being served stale results.

- Caching can be controlled using several HTTP headers: **`Date, Last Modified, Cache-Control, Expires, Age`**

X.1.9 Documenting a RESTful Service

- RESTful services do not necessarily require a document to help clients discover them (but they usually do so). Due to URIs, links, and a uniform interface, it is quite simple to discover RESTful services at runtime.

- A client can simply know the base address of the service and from there it can discover the service on its own.

X.2 Using a RESTful API with Python

- Many Restful web services API are available today. Just to mention a few :
  - With the **Facebook API**, programmers can integrate a more personalized social experience on their own websites, including the ability to “Like” and share pages.
  - With the **Twitter API** programmers have the ability to send and read tweets.
  - The **YouTube API** enables people to integrate the video content and functionality however they want. YouTube provides both a **YouTube Data API** and a **YouTube Player API**, allowing programmers great flexibility and control over YouTube’s functionality and content.
  - The **Flickr API** allows for development of different ways to share and organize photos, including integration with other social apps.
  - The **Dropbox API** grants access to Dropbox data from other business applications for easy revisions, file sharing, search, even restoration of previous file versions.
  - Developers can use the **AccuWeather API** to add weather alerts, daily and hourly forecasts, current conditions, and other weather-related functions to their applications.
  - ...

- Some of these API are free, and some are not. Most of them ask you to obtain and use API-Keys for API-requests.

X.2.1 What do we use to communicate with these APIs?

- To work with HTTP Python includes an interesting module called **`urllib.request`** but working with it can become cumbersome because it is quite « low level ».

- To develop REST clients program it is more convenient to use the module **`requests`** which simplifies the common use cases.

Note: To install the module requests, run the command: pip install requests

- Here is a code distinguishing `urllib.request` and `requests` :

```
import urllib.request as urllib2

gh_url = 'https://api.github.com'

req = urllib2.Request(gh_url)

password_manager = urllib2.HTTPPasswordMgrWithDefaultRealm()
password_manager.add_password(None, gh_url, 'uname', 'passwd')

auth_manager = urllib2.HTTPBasicAuthHandler(password_manager)
opener = urllib2.build_opener(auth_manager)
urllib2.install_opener(opener)
handler = urllib2.urlopen(req)
print(handler.getcode())
print(handler.getheader('content-type'))
```

```
import requests

r   =   requests.get('https://api.github.com',   auth=('aname',
'passwd'))

print(r.status_code)
print(r.headers['content-type'])
```

X.2.2 A first Request

- JSONPlaceholder is a free online REST service that you can use whenever you need some fake data. This service will be used in the following examples.

```
import requests

route="https://jsonplaceholder.typicode.com"
r = requests.get(route+'/users/1')

print(r.status_code)
print(r.headers['content-type'])
print(r.text)
```

- **`get()`** returns a response object using which we can get all the information needed about the service response to our `get()` request.

- `requests` will automatically decode content from the server. The text encoding guessed by `requests` is used when you access `r.text`. You can find out what encoding `requests` is using, and change it, using the `r.encoding` property.

- With the `requests` module all forms of HTTP request are as obvious :
  `requests.post(…)`
  `requests.put(…)`
  `requests.delete(…)`
  `requests.head(…)`
  `requests.options(…)`

- `requests` will automatically decode content from the server. The text encoding guessed by `requests` is used when you access `r.text`.

- You can find out what encoding `requests` is using, and change it, using the **`encoding`** property.

- The response objects includes a built-in JSON decoder **`json()`**.

```
import requests

route="https://jsonplaceholder.typicode.com"
r = requests.get(route+'/users/1')

r.encoding = 'ISO-8859-1'
print(r.status_code)
print(r.headers['content-type'])
print(r.json())
print(r.encoding) #ISO-8859-1
```

X.2.3 Passing parameters with URLs

- You often need to pass parameters to a RESTful web service.
  If you were constructing the URL by hand, this data would be given as key/value pairs in the URL after a question mark, e.g. https://jsonplaceholder.typicode.com/posts?userId=2

- requests help you provide the parameters easily as a `dict` provided via the `post()` **`params`** argument.

- You can then see that the URL has been correctly encoded by printing it (property **`url`**).

```
import requests
route="https://jsonplaceholder.typicode.com"
par={'userId': 2}
r = requests.get(route+"/posts", params=par)
print(r.text)
print(r.url)
```

X.2.4 Custom Headers

- If you'd like to add HTTP headers to a request, simply pass in a `dict` to the **`headers`** parameter.

- For example, we didn't specify our `content-type` in the previous example:

```
import requests
import json

route="https://jsonplaceholder.typicode.com"
par={"userId": 33, "name": "James Stewart",
     "username": "JStewart", "email": "j.stewart@gmail.com"}
headers={'content-type': 'application/json'}

r = requests.post(route+"/users",
                  data=json.dumps(par), headers=headers)
print(r.text)
```

- In the same way, most APIs require ***access token*** for requesting data. The access token needs to be added to HTTP headers.

```
headers = {'Authorization': 'token {}'.format(token)}
r = requests.get(url, headers=headers)
```

X.2.5 Response Status Codes

- We can check the status codes for the response using the property **`status_code`**
- If we receive an error code like 4XX or 5XX, we can raise it with the response method **`raise_for_status()` (**`raise_for_status()` returns `None` for status_code 200).

X.2.6 Response Headers

- We can view the server's response headers using a Python the dictionary **`headers`**:

```
r.headers
{
    'content-encoding': 'gzip',
    'transfer-encoding': 'chunked',
    …
}
```

- Many RESTful services use **`Link`** headers for pagination in their API. `requests` will automatically parse these link headers and make them easily consumable:

```
r.links["next"]
r.links["last"]
```

X.2.7 Timeouts

- You can tell requests to stop waiting for a response after a given number of seconds with the **`timeout`** parameter: `requests.get(url, timeout=0.001)`

X.2.8 Errors and Exception

- In case of network problem (e.g. DNS failure, refused connection, etc), requests will raise a **`ConnectionError`** exception.
- In the (rare) event of an invalid HTTP response, requests will raise an **`HTTPError`** exception.
- If a request times out, a **`Timeout`** exception is raised.

- If a request exceeds the configured number of maximum redirections, a **TooManyRedirects** exception is raised.

- All exceptions that Requests explicitly raises inherit from requests.exceptions.**RequestException**.

X.2.9 Payload

- The payload contains the data to be sent on the requests. Quite often the payload is represented by a JSON object.

- The **json** library in Python helps in creating a JSON object.

- After having set the header 'Content-type' to 'application/json', you can use the **data** parameter of the post() function.

- An another option is to directly provides a dict to the **json** parameter of the post() function.

X.2.10 Authorization

- The requests module supports various forms of authentication, which includes Basic, Digest Authentication, OAuth and others. The value for authentication can be passed using the **auth** parameter of the requests functions.

```
import requests
from requests.auth import HTTPBasicAuth

r = requests.get('https://api.github.com',
               auth=HTTPBasicAuth('alogname', 'apassword'))

print(r.status_code)
print(r.headers['content-type'])
```

- The auth argument can take any function, so if you want to define your own custom authentication and pass it to auth.

X.3 How to setup a RESTful API with Python (a short introduction to Flask)

- **Flask** is a simple, yet very powerful Python web framework.

- Building a RESTful web services with Flask is surprisingly simple: the following paragraph will try to demonstrate that.

Note: an alternative to Flask is to use **`Django`** (a popular python web framework), but Django is more complicated to learn and is heavier then Flask.

Note: instead of using a python script and, for instance, the `requests` module, you can use the **`curl`** (https://curl.haxx.se/) command to quickly make test requests to a Web Service.

X.3.1 A first complete example

- Let's begin by making a complete application that responds to requests at the following route: /, /articles and /articles/:id.

```
from flask import Flask, url_for
app = Flask(__name__)

@app.route('/')
def api_root():
    return 'Welcome'

@app.route('/articles')
def api_articles():
    return 'List of ' + url_for('api_articles')

@app.route('/articles/<articleid>')
def api_article(articleid):
    return 'You are reading ' + articleid

if __name__ == '__main__':
    app.run()
```

- You can use curl to make a request using:

```
curl http://127.0.0.1:5000/
```

- The `@app.route('/')` defines a **route**, which is mapped to a URL provided by the Web Service client (`curl` here).

- *`api_root()`* is the function that will answer to that route: it is a "**endpoint**".

- What happens in the end? Every time we hit the root (/) of our Web Service we will get the message "Welcome"!

- In this first example, other routes and endpoints do exist: `/articles` and `/articles/<article_id>`.
- `<articleid>` means we are expecting to receive a dynamic argument in the URL.
- We can indicate explicitly the type of argument expected, for instance:

  ```
  @app.route('/articles/<int:articleid>')
  @app.route('/articles/<float:articleid>')
  @app.route('/articles/<path:articleid>')
  ```

- The default is type `string` which accepts any text without slashes.

Note: **`url_for()`** generates a URL to the given endpoint with the method provided.

Note: Activating pretty (HTML) debug messages during development can be done simply by passing debug argument to the `run()` method: `app.run(debug=True)`

X.3.2 Endpoints methods

- By default, the endpoint will accept request using the HTTP GET method. If you want other methods to be accepted, you can use the following syntax:

```
@app.route('/echo', methods = ['GET', 'POST', 'PATCH', 'PUT',
'DELETE'])
def api_echo():
    if request.method == 'GET':
        return "ECHO: GET\n"

    elif request.method == 'POST':
        return "ECHO: POST\n"

    elif request.method == 'PATCH':
        return "ECHO: PACTH\n"

    elif request.method == 'PUT':
        return "ECHO: PUT\n"

    elif request.method == 'DELETE':
        return "ECHO: DELETE"
```

- To curl the **-X** option can be used to specify the request type:

  ```
  curl -X PUT http://127.0.0.1:5000/echo
  ```

X.3.3 The incoming request

- To access incoming request data, an endpoint can use the global **`request`** object.
- Flask parses incoming request data for you and gives you access to it through that global object.
- The `request` object provides attributes that help to handle the request. Here is a quick overview of the most important ones:

`form`: a kind of dict with the parsed form data from POST or PUT requests.

`args`: a kind of dict with the parsed contents of the query string (the part in the URL after the question mark).

`values`: a kind of dict with the contents of both `form` and `args`.

`cookies`: a dict with the contents of all cookies transmitted with the request.

`stream`: if the incoming form data was not encoded with a known mimetype the data is stored unmodified in this stream for consumption (the stream only returns the data once).

`headers`: the incoming request headers as a dictionary like object.

`data`: contains the incoming request data as string in case it came with a mimetype Flask does not handle.

`files`: a kind of dict with files uploaded as part of a POST or PUT request. Each file is stored as `FileStorage` object. It basically behaves like a standard file object you know from Python, with the difference that it also has a `save()` function that can store the file on the filesystem.

`method`: the current request method (POST, GET etc.)

`is_json`: indicates if this request is JSON or not (mimetype application/json or application/*+json).

`json`: if the mimetype is application/json this will contain the parsed JSON data. The **`get_json()`** method should be used instead.

```
from flask import json

@app.route('/messages', methods = ['POST'])
def api_message():
     if request.headers['Content-Type'] == 'text/plain':
          return "Text Message: " + request.data
     elif          request.headers['Content-Type']          ==
'application/json':
          return "JSON Message: " + json.dumps(request.json)
     elif          request.headers['Content-Type']          ==
'application/octet-stream':
          f = open('./binary', 'wb')
          f.write(request.data)
          f.close()
          return "Binary message written!"

     else:
          return "415 Unsupported Media Type ;)"
```

- To specify the content type with curl:

  ```
  curl -H "Content-type: application/json" \
  -X        POST       http://127.0.0.1:5000/messages       -d
  "{\"message\":\"Hello Data\"}"
  ```

- To send a file with curl:

  ```
  curl -H "Content-type: application/octet-stream" \
  -X    POST    http://127.0.0.1:5000/messages    --data-binary
  @image.png
  ```

- Also note that Flask can handle files POSTed via an HTML form using **`request.files`** and `curl` can simulate that behavior with the **-F** flag.

X.3.4 Responses

- Responses are handled by Flask's **`Response`** class:

```
Response(response=None, status=None, headers=None,
mimetype=None, content_type=None)
```

- The following attributes are available:

`headers`: an object representing the response headers.

`status`: a string with a response status.

`status_code`: the response status as integer.

`mimetype`: the mimetype (content type without charset etc.)

`set_cookie()`: sets a cookie.

```
from flask import Response

@app.route('/hello', methods = ['GET'])
def api_hello():
    data = {
        'hello'  : 'world',
        'number' : 3
    }
    js = json.dumps(data)

    resp = Response(js, status=200,
                         mimetype='application/json')
    resp.headers['Link'] = 'http://www.epfl.ch'

    return resp
```

- To view the response HTTP headers using curl, specify the **-i** option:

```
curl -i http://127.0.0.1:5000/hello
```

- The previous example can be further simplified by using a Flask convenience method for generating JSON responses. You can replace:

```
resp = Response(js, status=200, mimetype='application/json')
```

with:

```
from flask import jsonify
resp = jsonify(data)
resp.status_code = 200
```

X.3.5 Status Codes & Errors

- 200 is the default status code reply for GET requests, in both of the previous examples, specifying it was just for the sake of illustration.
- There are certain cases where overriding the defaults is necessary. Default Flask error messages can be overwritten using either the **`@error_handler`** decorator:

```
@app.errorhandler(404)
def not_found(error=None):
    message = {
            'status': 404,
            'message': 'Not Found: ' + request.url,
    }
    resp = jsonify(message)
    resp.status_code = 404

    return resp

@app.route('/users/<userid>', methods = ['GET'])
def api_users(userid):
    users = {'1':'john', '2':'steve', '3':'bill'}

    if userid in users:
        return jsonify({userid:users[userid]})
    else:
        return not_found()
```

XI INDEX