

Gustafson's law

Gustafson's law is based on the following considerations:

- ▶ While increasing the dimension of a problem, its sequential parts remain constant
- ▶ While increasing the number of processors, the work required on each of them still remains the same

This states that $S(P) = P - \alpha (P - 1)$, where P is the number of processors, S is the speedup, and α is the non-parallelizable fraction of any parallel process. This is in contrast to Amdahl's law, which takes the single-process execution time to be the fixed quantity and compares it to a shrinking per process parallel execution time. Thus, Amdahl's law is based on the assumption of a fixed problem size; it assumes that the overall workload of a program does not change with respect to the machine size (that is, the number of processors). Gustafson's law addresses the deficiency of Amdahl's law, which does not take into account the total number of computing resources involved in solving a task. It suggests that the best way to set the time allowed for the solution of a parallel problem is to consider all the computing resources and on the basis of this information, it fixes the problem.

Introducing Python

Python is a powerful, dynamic, and interpreted programming language that is used in a wide variety of applications. Some of its features include:

- ▶ A clear and readable syntax
- ▶ A very extensive standard library, where through additional software modules, we can add data types, functions, and objects
- ▶ Easy-to-learn rapid development and debugging; the development of Python code in Python can be up to 10 times faster than the C/C++ code
- ▶ Exception-based error handling
- ▶ A strong introspection functionality
- ▶ Richness of documentation and software community

Python can be seen as a glue language. Using Python, better applications can be developed because different kinds of programmers can work together on a project. For example, when building a scientific application, C/C++ programmers can implement efficient numerical algorithms, while scientists on the same project can write Python programs that test and use those algorithms. Scientists don't have to learn a low-level programming language and a C/C++ programmer doesn't need to understand the science involved.



You can read more about this from <https://www.python.org/doc/essays/omg-darpa-mcc-position>.

Getting ready

Python can be downloaded from <https://www.python.org/downloads/>.

Although you can create Python programs with Notepad or TextEdit, you'll notice that it's much easier to read and write code using an **Integrated Development Environment (IDE)**.

There are many IDEs that are designated specifically for Python, including IDLE (<http://www.python.org/idle>), PyCharm (<https://www.jetbrains.com/pycharm/>), and Sublime Text, (<http://www.sublimetext.com/>).

How to do it...

Let's take a look at some examples of the very basic code to get an idea of the features of Python. Remember that the symbol `>>>` denotes the Python shell:

- ▶ Operations with integers:

```
>>> # This is a comment
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Only for this first example, we will see how the code appears in the Python shell:

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> #This is a comment
>>> width = 20
>>> height = 5*9
>>> width * height
900
>>>
```

Let's see the other basic examples:

- ▶ Complex numbers:

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
>>> abs(a) # sqrt(a.real**2 + a.imag**2)
5.0
```

- ▶ Strings manipulation:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[-1] # The last character
'A'
```

- ▶ Defining lists:

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> len(a)
4
```

- ▶ The while loop:

```
# Fibonacci series:
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

- ▶ The if command:

First we use the `input()` statement to insert an integer:

```
>>>x = int(input("Please enter an integer here: "))
Please enter an integer here:
```

Then we implement the `if` condition on the number inserted:

```
>>>if x < 0:
...     print ('the number is negative')
...elif x == 0:
...     print ('the number is zero')
...elif x == 1:
...     print ('the number is one')
...else:
...     print ('More')
...
```

- ▶ The for loop:

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print (x, len(x))
...
cat 3
window 6
defenestrate 12
```

▶ Defining functions:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while b < n:
...         print (b),
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

▶ Importing modules:

```
>>> import math
>>> math.sin(1)
0.8414709848078965
```

```
>>> from math import *
>>> log(1)
0.0
```

▶ Defining classes:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Python in a parallel world

To be an interpreted language, Python is fast, and if speed is critical, it easily interfaces with extensions written in faster languages, such as C or C++. A common way of using Python is to use it for the high-level logic of a program; the Python interpreter is written in C and is known as CPython. The interpreter translates the Python code in an intermediate language called Python bytecode, which is analogous to an assembly language, but contains a high level of instruction. While a Python program runs, the so-called evaluation loop translates Python bytecode into machine-specific operations. The use of interpreter has advantages in code programming and debugging, but the speed of a program could be a problem. A first solution is provided by third-party packages, where a programmer writes a C module and then imports it from Python. Another solution is the use of a Just-in-Time Python compiler, which is an alternative to CPython, for example, the PyPy implementation optimizes code generation and the speed of a Python program. In this book, we will examine a third approach to the problem; in fact, Python provides ad hoc modules that could benefit from parallelism. The description of many of these modules, in which the parallel programming paradigm falls, will be discussed in subsequent chapters.

However, in this chapter, we will introduce the two fundamental concepts of threads and processes and how they are addressed in the Python programming language.

Introducing processes and threads

A process is an executing instance of an application, for example, double-clicking on the Internet browser icon on the desktop will start a process than runs the browser. A thread is an active flow of control that can be activated in parallel with other threads within the same process. The term "flow control" means a sequential execution of machine instructions. Also, a process can contain multiple threads, so starting the browser, the operating system creates a process and begins executing the primary threads of that process. Each thread can execute a set of instructions (typically, a function) independently and in parallel with other processes or threads. However, being the different active threads within the same process, they share space addressing and then the data structures. A thread is sometimes called a lightweight process because it shares many characteristics of a process, in particular, the characteristics of being a sequential flow of control that is executed in parallel with other control flows that are sequential. The term "light" is intended to indicate that the implementation of a thread is less onerous than that of a real process. However, unlike the processes, multiple threads may share many resources, in particular, space addressing and then the data structures.

Let's recap:

- ▶ A process can consist of multiple parallel threads.
- ▶ Normally, the creation and management of a thread by the operating system is less expensive in terms of CPU's resources than the creation and management of a process. Threads are used for small tasks, whereas processes are used for more heavyweight tasks—basically, the execution of applications.
- ▶ The threads of the same process share the address space and other resources, while processes are independent of each other.

Before examining in detail the features and functionality of Python modules for the management of parallelism via threads and processes, let's first look at how the Python programming language works with these two entities.

Start working with processes in Python

On common operating systems, each program runs in its own process. Usually, we start a program by double-clicking on the icon's program or selecting it from a menu. In this recipe, we simply demonstrate how to start a single new program from inside a Python program. A process has its own space address, data stack, and other auxiliary data to keep track of the execution; the OS manages the execution of all processes, managing the access to the computational resources of the system via a scheduling procedure.

Getting ready

In this first Python application, you'll simply get the Python language installed.



Refer to <https://www.python.org/> to get the latest version of Python.

How to do it...

To execute this first example, we need to type the following two programs:

- ▶ `called_Process.py`
- ▶ `calling_Process.py`

You can use the Python IDE (3.3.0) to edit these files:

The code for the `called_Process.py` file is as shown:

```
print ("Hello Python Parallel Cookbook!!")
closeInput = raw_input("Press ENTER to exit")
print "Closing calledProcess"
```

The code for the `calling_Process.py` file is as shown:

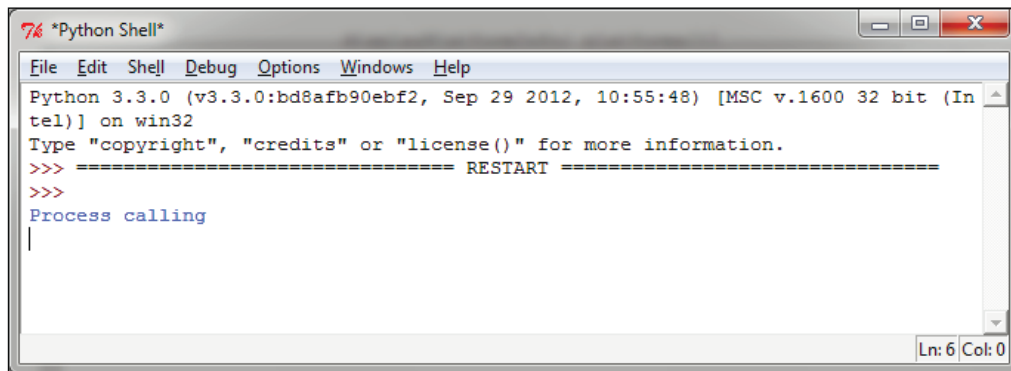
```
##The following modules must be imported
import os
import sys

##this is the code to execute
program = "python"
print("Process calling")
arguments = ["called_Process.py"]

##we call the called_Process.py script
os.execvp(program, (program,) + tuple(arguments))
print("Good Bye!!")
```

To run the example, open the `calling_Process.py` program with the Python IDE and then press the `F5` button on the keyboard.

You will see the following output in the Python shell:

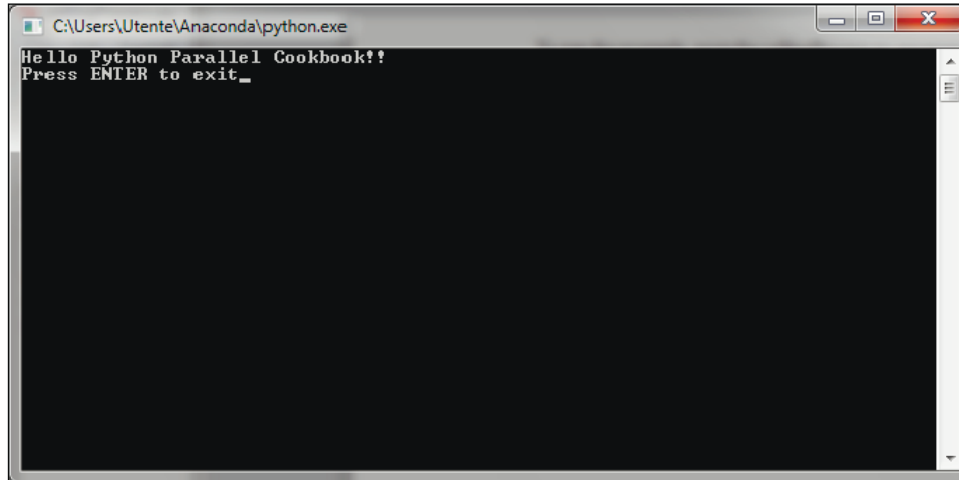


The screenshot shows a window titled "*Python Shell*" with a menu bar (File, Edit, Shell, Debug, Options, Windows, Help). The shell displays the following text:

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Process calling
|
```

The status bar at the bottom right indicates "Ln: 6 Col: 0".

At same time, the OS prompt displays the following:

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Users\Utente\Anaconda\python.exe'. The window content displays the text 'Hello Python Parallel Cookbook!' followed by 'Press ENTER to exit_'. The rest of the window is black, indicating that the program has finished execution and the prompt is waiting for input.

We have two processes running to close the OS prompt; simply press the *Enter* button on the keyboard to do so.

How it works...

In the preceding example, the `execvp` function starts a new process, replacing the current one. Note that the "Good Bye" message is never printed. Instead, it searches for the program called `_Process.py` along the standard path, passes the contents of the second argument tuple as individual arguments to that program, and runs it with the current set of environment variables. The instruction `input()` in `called_Process.py` is only used to manage the closure of OS prompt. In the recipe dedicated to process-based parallelism, we will finally see how to manage a parallel execution of more processes via the multiprocessing Python module.

Start working with threads in Python

As mentioned briefly in the previous section, thread-based parallelism is the standard way of writing parallel programs. However, the Python interpreter is not fully thread-safe. In order to support multithreaded Python programs, a global lock called the **Global Interpreter Lock (GIL)** is used. This means that only one thread can execute the Python code at the same time; Python automatically switches to the next thread after a short period of time or when a thread does something that may take a while. The GIL is not enough to avoid problems in your own programs. Although, if multiple threads attempt to access the same data object, it may end up in an inconsistent state.

In this recipe, we simply show you how to create a single thread inside a Python program.

How to do it...

To execute this first example, we need the program `helloPythonWithThreads.py`:

```
## To use threads you need import Thread using the following code:
from threading import Thread

##Also we use the sleep function to make the thread "sleep"
from time import sleep

## To create a thread in Python you'll want to make your class work as a
thread.
## For this, you should subclass your class from the Thread class
class Cookbook(Thread):
    def __init__(self):
        Thread.__init__(self)
        self.message = "Hello Parallel Python Cookbook!!\n"

##this method prints only the message
    def print_message(self):
        print (self.message)

##The run method prints ten times the message
    def run(self):
        print ("Thread Starting\n")
        x=0
        while (x < 10):
            self.print_message()
            sleep(2)
            x += 1
        print ("Thread Ended\n")

#start the main process
print ("Process Started")
```

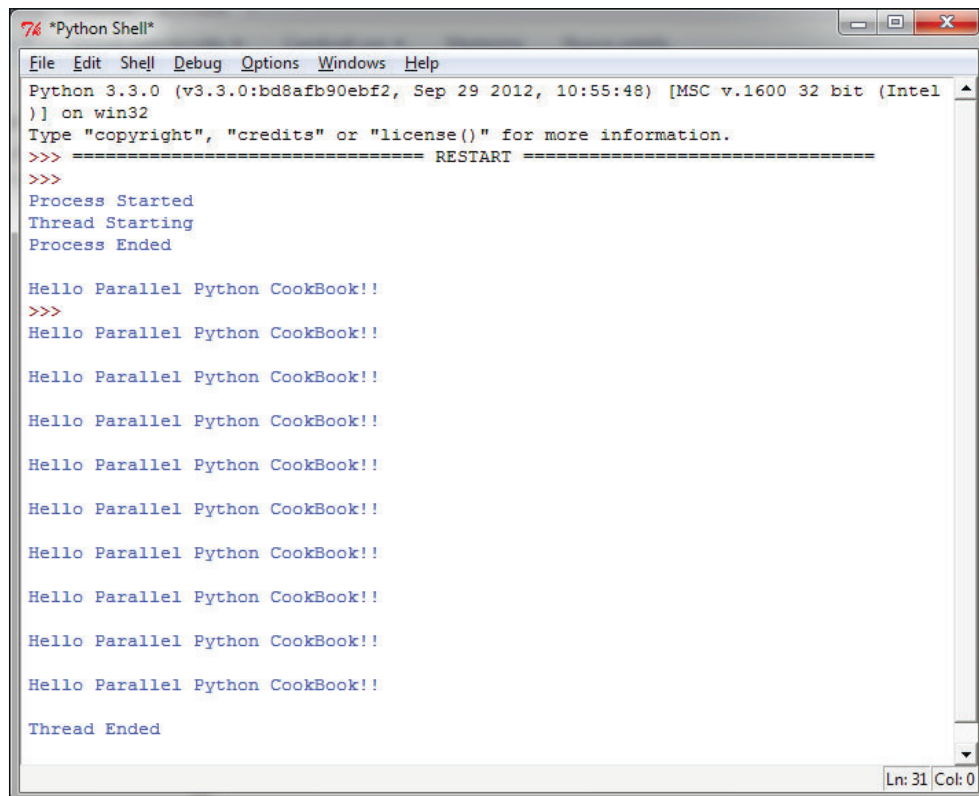
```
# create an instance of the HelloWorld class
hello_Python = CookBook()

# print the message...starting the thread
hello_Python.start()

#end the main process
print ("Process Ended")
```

To run the example, open the `calling_Process.py` program with the Python IDE and then press the `F5` button on the keyboard.

You will see the following output in the Python shell:



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8af90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel
)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Process Started
Thread Starting
Process Ended

Hello Parallel Python CookBook!!
>>>
Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Thread Ended
Ln: 31 Col: 0
```

How it works...

While the main program has reached the end, the thread continues printing its message every two seconds. This example demonstrates what threads are—a subtask doing something in a parent process.

A key point to make when using threads is that you must always make sure that you never leave any thread running in the background. This is very bad programming and can cause you all sorts of pain when you work on bigger applications.