Python has long been popular among developers, but the venerable programming language seems to be having a moment. After years of playing second fiddle to Java, some sources now rate Python as the world's most popular programming language.

Don't miss: Java vs. Python: Which Programming Language Is Best for You?

In fact, IEEE Spectrum even has Python extending its lead over C, C++, and Java! And they suggest a couple of reasons why: First, as modern microcontrollers continue to grow in power, they have the power to host a Python interpreter, allowing Python to be listed as an embedded language, spreading its reach. Second, Python may be taking advantage of a drop in R usage, further cementing Python's dominance in handling statistics and big data applications.

Python's flexibility FTW

Among the biggest reasons that Python is so hot are its relative simplicity and incredible flexibility. Few programming languages can match Python's ability to conform to your particular coding style, rather than forcing you to code in a certain way.

While Python does require you to adhere to certain conventions, such as adding white space (see the Pep-8 style guide) in order to make it easy to read and understand, it still retains the flexibility to let developers code in ways that feel natural to them. That comfort level can significantly improve developer efficiency and reduce the potential for errors.

This flexibility has another key benefit: Rather than enforcing use of a particular coding style, it lets more advanced developers use the style they feel is best suited to solving a particular problem.

There are four main Python coding styles: **imperative, functional, object-oriented,** and **procedural.** (Some people combine imperative and functional coding styles while others view them as completely separate styles.) You may or may not agree that all four forms are valid or even useful—but nevertheless Python makes them all available. Let's take a look at the pros and cons of each approach as well as some examples.

A brief overview of the four Python coding styles

1. **Functional:** Every statement is treated as a mathematical equation and any forms of state or mutable data are avoided. The main advantage of this approach is that it lends itself well to parallel processing because there is no state to consider. Many developers prefer this coding style for recursion and for lambda

calculus. (Note that Python's implementation of functional programming deviates from the standard—read, is impure— because it's possible to maintain state and create side effects if you're not careful. If you need a pure functional programming implementation, Haskell may be a better choice.)

2. **Imperative:** Computation is performed as a direct change to program state. This style is especially useful when manipulating data structures and produces elegant yet simple code. Python fully implements this paradigm.
3. **Object-oriented:** Relies on data fields that are treated as objects and manipulated only through prescribed methods. Python doesn't fully support this paradigm because it can't implement features such as data hiding (encapsulation), which many believe is a primary requirement of the object-oriented programming paradigm. This coding style also favors code reuse.
4. **Procedural:** Tasks are treated as step-by-step iterations where common tasks are placed in functions that are called as needed. This coding style favors iteration, sequencing, selection, and modularization. Python excels in implementing this particular paradigm.

Four styles, one example

Normally, you'd choose a particular coding style to meet a specific need, but using a common problem as an example makes it easier to compare the different styles. Our example is designed to compute the sum of the following list:

my_list = [1, 2, 3, 4, 5]

1. Using the functional coding style

The functional coding style treats everything like a math equation. The two most common ways to compute the sum of **my_list** would be to use a local function or a lambda expression. Here's how you'd do it with a local function in Python 3.6:

```
import functools
my_list = [1, 2, 3, 4, 5]
def add_it(x, y):
    return (x + y)
sum = functools.reduce(add_it, my_list)
print(sum)
```

The functools package provides access to higher-order functions for data manipulation. However, you don't always use it to perform functional programming with Python. Here's a simple example using **my_list** with a lambda function:

```
square = lambda x: x**2
double = lambda x: x + x
print(list(map(square, my_list)))
print(list(map(double, my_list)))
```

As you can see, the lambda expression is simpler (or, at least, shorter) than a similar procedural approach. Here's a lambda function version of the **functools.reduce()** call:

```
import functools
my_list = [1, 2, 3, 4, 5]
sum = functools.reduce(lambda x, y: x + y, my_list)
print(sum)
```

2. Using the imperative coding style

In imperative programming, you focus on *how* a program operates. Programs change state information as needed in order to achieve a goal. Here's an example using **my_list**:

```
sum = 0
for x in my_list:
    sum += x
print(sum)
```

Unlike the previous examples, the value of **sum** changes with each iteration of the loop. As a result, **sum** has state. When a variable has state, something must maintain that state, which means that the variable is tied to a specific processor. Imperative coding works on simple applications, but code executes too slowly for optimal results on complex data science applications.

3. Using the object-oriented coding style

Object-oriented coding is all about increasing an application's ability to reuse code and make it easier to understand. The encapsulation that object-orientation provides allows developers to treat code as a black box. Using object-orientation features like inheritance make it easier to expand the functionality of existing code. Here is the **my_list** example in object-oriented form:

```
class ChangeList(object):
    def __init__(self, any_list):
        self.any_list = any_list
    def do_add(self):
      self.sum = sum(self.any_list)
create_sum = ChangeList(my_list)
create_sum.do_add()
print(create_sum.sum)
```

In this case, **create_sum** is an instance of **ChangeList**. The inner workings of **ChangeList** don't matter to the person using it. All that really matters is that you can create an instance using a list and then call the **do_add()** method to output the sum of the list elements. Because the inner workings are hidden, the overall application is easier to understand.

4. Using the procedural coding style

The procedural style relies on procedure calls to create modularized code. This approach simplifies your application code by breaking it into small pieces that a developer can view easily. Even though procedural coding is an older form of application development, it's still a viable approach for tasks that lend themselves to step-by-step execution. Here's an example of the procedural coding style using **my_list**:

```
def do_add(any_list):
    sum = 0
    for x in any_list:
        sum += x
    return sum
print(do_add(my_list))
```

The use of a function, **do_add()**, simplifies the overall code in this case. The execution is still systematic, but the code is easier to understand because it's broken into chunks. However, this code suffers from the same issues as the imperative paradigm in that the use of state limits execution options, which means that this approach may not use hardware efficiently when tackling complex problems.

Choosing a coding style

Developers will differ on the coding styles—everyone has an opinion on which style is best. (I've listed the four styles in my personal order of preference.)

The amazing thing about Python is that it lets you choose a programming paradigm that works best for you in a given situation. It's even possible to mix and match paradigms within a Python application as long as you remember to keep packages limited to inputs and outputs (keeping the code modular). There are no rules that say you can't combine styles as needed. Python doesn't stop in the middle of interpreting your application and display a style error when you mix styles.

If you're not sure which coding style will work best for a given task, try several of the styles to determine which one helps you solve the problem fastest and with the least convoluted code. Again, you may find that a single style doesn't truly solve the problem and you want to incorporate several styles on the same application.

Finally, when you find an efficient solution to a problem, be sure to clearly document it so that you—or another programmer—can work on it later without having to reinvent the wheel. That's especially important when mixing programming styles because an increase in flexibility can sometimes decrease clarity.

Done right, today's problem solved can become tomorrow's time-saving template in terms of choosing the right Python programming style.