Conditional Statements in Python

Table of Contents

Conditional Statements in Python (if/elif/else)

From the previous tutorials in this series, you now have quite a bit of Python code under your belt. Everything you have seen so far has consisted of sequential execution, in which statements are always performed one after the next, in exactly the order specified.

But the world is often more complicated than that. Frequently, a program needs to skip over some statements, execute a series of statements repetitively, or choose between alternate sets of statements to execute.

That is where control structures come in. A control structure directs the order of execution of the statements in a program (referred to as the program's control flow).

Here's what you'll learn in this tutorial: You'll encounter your first Python control structure, the `if` statement.

In the real world, we commonly must evaluate information around us and then choose one course of action or another based on what we observe:

If the weather is nice, then I'll mow the lawn. (It's implied that if the weather isn't nice, then I won't mow the lawn.)
In a Python program, the `if` statement is how you perform this sort of decision-making. It allows for conditional execution of a statement or group of statements based on the value of an expression.

The outline of this tutorial is as follows:

- First, you'll get a quick overview of the `if` statement in its simplest form.
- Next, using the `if` statement as a model, you'll see why control structures require some mechanism for grouping statements together into compound statements or blocks. You'll learn how this is done in Python.
- Lastly, you'll tie it all together and learn how to write complex decision-making code.

Ready? Here we go!

Introduction to the `if` Statement

We'll start by looking at the most basic type of `if` statement. In its simplest form, it looks like this:

```
if <expr>:
    <statement>
```

In the form shown above:

- `<expr>` is an expression evaluated in a Boolean context, as discussed in the section on Logical Operators in the Operators and Expressions in Python tutorial.
- `<statement>` is a valid Python statement, which must be indented. (You will see why very soon.)

If `<expr>` is true (evaluates to a value that is “truthy”), then `<statement>` is executed. If `<expr>` is false, then `<statement>` is skipped over and not executed.

Note that the colon (`:`) following `<expr>` is required. Some programming languages require `<expr>` to be enclosed in parentheses, but Python does not.

Here are several examples of this type of `if` statement:

```
>>>
>>> x = 0
>>> y = 5

>>> if x < y:                                   # Truthy
...     print('yes')
...
yes
>>> if y < x:                                   # Falsy
...     print('yes')
...

>>> if x:                                       # Falsy
...     print('yes')
...
>>> if y:                                       # Truthy
...     print('yes')
...
yes

>>> if x or y:                                  # Truthy
...     print('yes')
...
yes
>>> if x and y:                                 # Falsy
...     print('yes')
...

>>> if 'aul' in 'grault':                       # Truthy
...     print('yes')
...
yes
>>> if 'quux' in ['foo', 'bar', 'baz']:         # Falsy
...     print('yes')
```

```
...
```

Note: If you are trying these examples interactively in a REPL session, you'll find that, when you hit `Enter` after typing in the `print('yes')` statement, nothing happens.

Because this is a multiline statement, you need to hit `Enter` a second time to tell the interpreter that you're finished with it. This extra newline is not necessary in code executed from a script file.

Grouping Statements: Indentation and Blocks

So far, so good.

But let's say you want to evaluate a condition and then do more than one thing if it is true:

If the weather is nice, then I will:

- Mow the lawn
- Weed the garden
- Take the dog for a walk

(If the weather isn't nice, then I won't do any of these things.)
In all the examples shown above, each `if <expr>:` has been followed by only a single `<statement>`. There needs to be some way to say "If `<expr>` is true, do all of the following things."

The usual approach taken by most programming languages is to define a syntactic device that groups multiple statements into one compound statement or block. A block is regarded syntactically as a single entity. When it is the target of an `if` statement, and `<expr>` is true, then all the statements in the block are executed. If `<expr>` is false, then none of them are.

Virtually all programming languages provide the capability to define blocks, but they don't all provide it in the same way. Let's see how Python does it.

Python: It's All About the Indentation

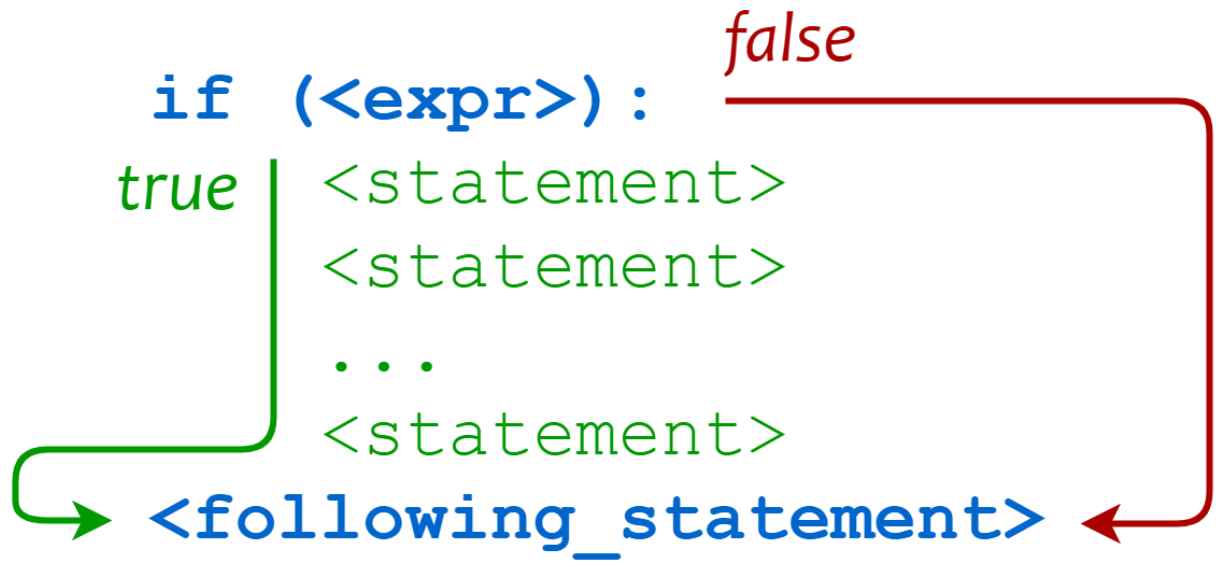
Python follows a convention known as the off-side rule, a term coined by British computer scientist Peter J. Landin. (The term is taken from the offside law in association football.) Languages that adhere to the off-side rule define blocks by indentation. Python is one of a relatively small set of off-side rule languages.

Recall from the previous tutorial on Python program structure that indentation has special significance in a Python program. Now you know why: indentation is used to define compound statements or blocks. In a Python program, contiguous statements that are indented to the same level are considered to be part of the same block.

Thus, a compound `if` statement in Python looks like this:

```
1if <expr>:
 2    <statement>
 3    <statement>
 4    ...
 5    <statement>
 6<following_statement>
```

Here, all the statements at the matching indentation level (lines 2 to 5) are considered part of the same block. The entire block is executed if `<expr>` is true, or skipped over if `<expr>` is false. Either way, execution proceeds with `<following_statement>` (line 6) afterward.



Python Compound if Statement

Notice that there is no token that denotes the end of the block. Rather, the end of the block is indicated by a line that is indented less than the lines of the block itself.

Note: In the Python documentation, a group of statements defined by indentation is often referred to as a suite. This tutorial series uses the terms block and suite interchangeably.

Consider this script file `foo.py`:

```
1if 'foo' in ['bar', 'baz', 'qux']:
 2    print('Expression was true')
 3    print('Executing statement in suite')
 4    print('...')
 5    print('Done.')
 6print('After conditional')
```

Running `foo.py` produces this output:

```
C:\Users\john\Documents>python foo.py
After conditional
```

The four `print()` statements on lines 2 to 5 are indented to the same level as one another. They constitute the block that would be executed if the condition

were true. But it is false, so all the statements in the block are skipped. After the end of the compound `if` statement has been reached (whether the statements in the block on lines 2 to 5 are executed or not), execution proceeds to the first statement having a lesser indentation level: the `print()` statement on line 6.

Blocks can be nested to arbitrary depth. Each indent defines a new block, and each outdent ends the preceding block. The resulting structure is straightforward, consistent, and intuitive.

Here is a more complicated script file called `blocks.py`:

```
# Does line execute?                          Yes     No
#                                             ---     --
if 'foo' in ['foo', 'bar', 'baz']:          #   x
    print('Outer condition is true')        #   x

    if 10 > 20:                             #   x
        print('Inner condition 1')          #         x

    print('Between inner conditions')       #   x

    if 10 < 20:                             #   x
        print('Inner condition 2')          #   x

    print('End of outer condition')         #   x
print('After outer condition')              #   x
```

The output generated when this script is run is shown below:

```
C:\Users\john\Documents>python blocks.py
Outer condition is true
Between inner conditions
Inner condition 2
End of outer condition
After outer condition
```

Note: In case you have been wondering, the off-side rule is the reason for the necessity of the extra newline when entering multiline statements in a REPL

session. The interpreter otherwise has no way to know that the last statement of the block has been entered.

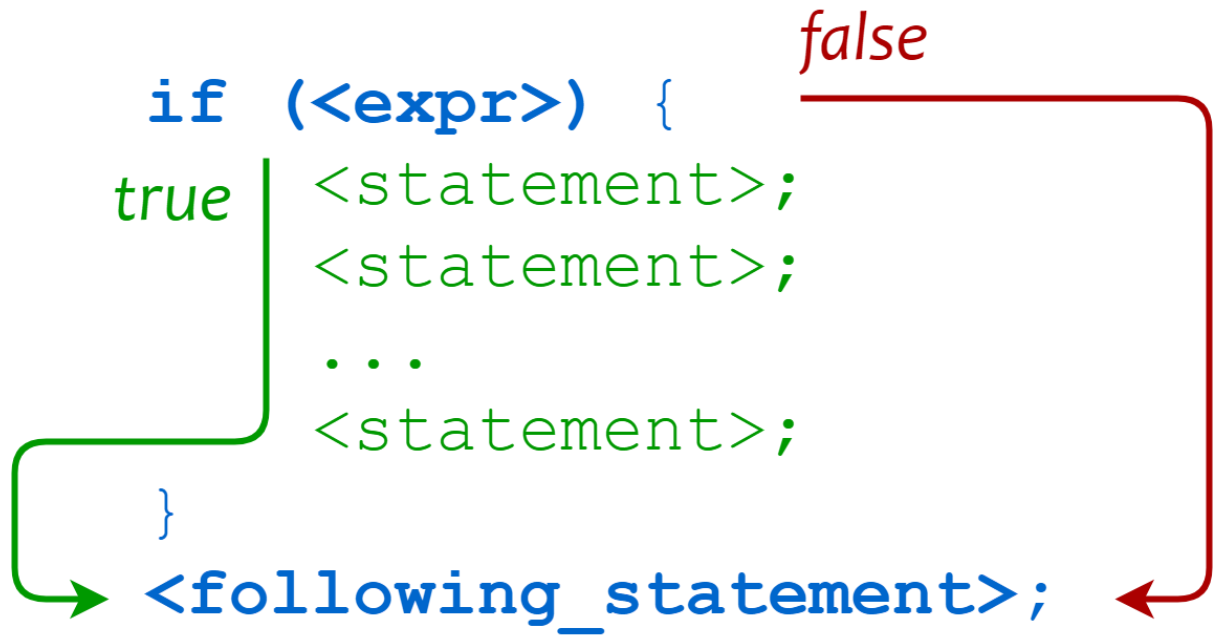
What Do Other Languages Do?

Perhaps you're curious what the alternatives are. How are blocks defined in languages that don't adhere to the off-side rule?

The tactic used by most programming languages is to designate special tokens that mark the start and end of a block. For example, in Perl blocks are defined with pairs of curly braces (`{}`) like this:

```
# (This is Perl, not Python)
if (<expr>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
<following_statement>;
```

C/C++, Java, and a whole host of other languages use curly braces in this way.



Compound if Statement in C/C++, Perl, and Java

Other languages, such as Algol and Pascal, use keywords `begin` and `end` to enclose blocks.

Which Is Better?

Better is in the eye of the beholder. On the whole, programmers tend to feel rather strongly about how they do things. Debate about the merits of the off-side rule can run pretty hot.

On the plus side:

- Python's use of indentation is clean, concise, and consistent.
- In programming languages that do not use the off-side rule, indentation of code is completely independent of block definition and code function. It's possible to write code that is indented in a manner that does not actually match how the code executes, thus creating a mistaken impression when a person just glances at it. This sort of mistake is virtually impossible to make in Python.
- Use of indentation to define blocks forces you to maintain code formatting standards you probably should be using anyway.

On the negative side:

- Many programmers don't like to be forced to do things a certain way. They tend to have strong opinions about what looks good and what doesn't, and they don't like to be shoehorned into a specific choice.
- Some editors insert a mix of space and tab characters to the left of indented lines, which makes it difficult for the Python interpreter to determine indentation levels. On the other hand, it is frequently possible to configure editors not to do this. It generally isn't considered desirable to have a mix of tabs and spaces in source code anyhow, no matter the language.

Like it or not, if you're programming in Python, you're stuck with the off-side rule. All control structures in Python use it, as you will see in several future tutorials.

For what it's worth, many programmers who have been used to languages with more traditional means of block definition have initially recoiled at Python's way but have gotten comfortable with it and have even grown to prefer it.

The `else` and `elif` Clauses

Now you know how to use an `if` statement to conditionally execute a single statement or a block of several statements. It's time to find out what else you can do.

Sometimes, you want to evaluate a condition and take one path if it is true but specify an alternative path if it is not. This is accomplished with an `else` clause:

```
if <expr>:
    <statement(s)>
else:
    <statement(s)>
```

If `<expr>` is true, the first suite is executed, and the second is skipped. If `<expr>` is false, the first suite is skipped and the second is executed. Either way, execution then resumes after the second suite. Both suites are defined by indentation, as described above.

In this example, `x` is less than `50`, so the first suite (lines 4 to 5) are executed, and the second suite (lines 7 to 8) are skipped:

```
>>>
1>>> x = 20
 2
 3>>> if x < 50:
 4...     print('(first suite)')
 5...     print('x is small')
 6... else:
 7...     print('(second suite)')
 8...     print('x is large')
 9...
10(first suite)
11x is small
```

Here, on the other hand, `x` is greater than `50`, so the first suite is passed over, and the second suite executed:

```
>>>
1>>> x = 120
 2>>>
 3>>> if x < 50:
 4...     print('(first suite)')
 5...     print('x is small')
 6... else:
 7...     print('(second suite)')
 8...     print('x is large')
 9...
10(second suite)
11x is large
```

There is also syntax for branching execution based on several alternatives. For this, use one or more `elif` (short for *else if*) clauses. Python evaluates each `<expr>` in turn and executes the suite corresponding to the first that is

true. If none of the expressions are true, and an `else` clause is specified, then its suite is executed:

```
if <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
    ...
else:
    <statement(s)>
```

An arbitrary number of `elif` clauses can be specified. The `else` clause is optional. If it is present, there can be only one, and it must be specified last:

```
>>>
>>> name = 'Joe'
>>> if name == 'Fred':
...     print('Hello Fred')
... elif name == 'Xander':
...     print('Hello Xander')
... elif name == 'Joe':
...     print('Hello Joe')
... elif name == 'Arnold':
...     print('Hello Arnold')
... else:
...     print("I don't know who you are!")
...
Hello Joe
```

At most, one of the code blocks specified will be executed. If an `else` clause isn't included, and all the conditions are false, then none of the blocks will be executed.

Note: Using a lengthy `if`/`elif`/`else` series can be a little inelegant, especially when the actions are simple statements like `print()`. In many cases, there may be a more Pythonic way to accomplish the same thing.

Here's one possible alternative to the example above using the `dict.get()` method:

```
>>>
>>> names = {
...     'Fred': 'Hello Fred',
...     'Xander': 'Hello Xander',
...     'Joe': 'Hello Joe',
...     'Arnold': 'Hello Arnold'
... }

>>> print(names.get('Joe', "I don't know who you are!"))
Hello Joe
>>> print(names.get('Rick', "I don't know who you are!"))
I don't know who you are!
```

Recall from the tutorial on Python dictionaries that the `dict.get()` method searches a dictionary for the specified key and returns the associated value if it is found, or the given default value if it isn't.

An `if` statement with `elif` clauses uses short-circuit evaluation, analogous to what you saw with the `and` and `or` operators. Once one of the expressions is found to be true and its block is executed, none of the remaining expressions are tested. This is demonstrated below:

```
>>>
>>> var  # Not defined
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    var
NameError: name 'var' is not defined

>>> if 'a' in 'bar':
...     print('foo')
... elif 1/0:
...     print("This won't happen")
... elif var:
...     print("This won't either")
...
foo
```

The second expression contains a division by zero, and the third references an undefined variable `var`. Either would raise an error, but neither is evaluated because the first condition specified is true.

One-Line `if` Statements

It is customary to write `if <expr>` on one line and `<statement>` indented on the following line like this:

```
if <expr>:
    <statement>
```

But it is permissible to write an entire `if` statement on one line. The following is functionally equivalent to the example above:

```
if <expr>: <statement>
```

There can even be more than one `<statement>` on the same line, separated by semicolons:

```
if <expr>: <statement_1>; <statement_2>; ...; <statement_n>
```

But what does this mean? There are two possible interpretations:

1. If `<expr>` is true, execute `<statement_1>`.
   Then, execute `<statement_2> ... <statement_n>` unconditionally, irrespective of whether `<expr>` is true or not.
2. If `<expr>` is true, execute all of `<statement_1> ... <statement_n>`.
   Otherwise, don't execute any of them.

Python takes the latter interpretation. The semicolon separating the `<statements>` has higher precedence than the colon following `<expr>`—in computer lingo, the semicolon is said to bind more tightly than the colon. Thus, the `<statements>` are treated as a suite, and either all of them are executed, or none of them are:

```
>>>
```

```
>>> if 'f' in 'foo': print('1'); print('2'); print('3')
...
1
2
3
>>> if 'z' in 'foo': print('1'); print('2'); print('3')
...
```

Multiple statements may be specified on the same line as an `elif` or `else` clause as well:

```
>>>
>>> x = 2
>>> if x == 1: print('foo'); print('bar'); print('baz')
... elif x == 2: print('qux'); print('quux')
... else: print('corge'); print('grault')
...
qux
quux

>>> x = 3
>>> if x == 1: print('foo'); print('bar'); print('baz')
... elif x == 2: print('qux'); print('quux')
... else: print('corge'); print('grault')
...
corge
grault
```

While all of this works, and the interpreter allows it, it is generally discouraged on the grounds that it leads to poor readability, particularly for complex `if` statements. PEP 8 specifically recommends against it.

As usual, it is somewhat a matter of taste. Most people would find the following more visually appealing and easier to understand at first glance than the example above:

```
>>>
>>> x = 3
>>> if x == 1:
...     print('foo')
...     print('bar')
```

```
...     print('baz')
... elif x == 2:
...     print('qux')
...     print('quux')
... else:
...     print('corge')
...     print('grault')
...
corge
grault
```

If an, if statement is simple enough, though, putting it all on one line, may be reasonable. Something like this probably wouldn't raise anyone's hackles too much:

```
debugging = True  # Set to True to turn debugging on.

    .
    .
    .

if debugging: print('About to call function foo()')
foo()
```

Conditional Expressions (Python's Ternary Operator)

Python supports one additional decision-making entity called a conditional expression. (It is also referred to as a conditional operator or ternary operator in various places in the Python documentation.) Conditional expressions were proposed for addition to the language in PEP 308 and green-lighted by Guido in 2005.

In its simplest form, the syntax of the conditional expression is as follows:

```
<expr1> if <conditional_expr> else <expr2>
```

This is different from the `if` statement forms listed above because it is not a control structure that directs the flow of program execution. It acts more like an operator that defines an expression. In the above example,

`<conditional_expr>` is evaluated first. If it is true, the expression evaluates to `<expr1>`. If it is false, the expression evaluates to `<expr2>`.

Notice the non-obvious order: the middle expression is evaluated first, and based on that result, one of the expressions on the ends is returned. Here are some examples that will hopefully help clarify:

```
>>>
>>> raining = False
>>> print("Let's go to the", 'beach' if not raining else 'library')
Let's go to the beach
>>> raining = True
>>> print("Let's go to the", 'beach' if not raining else 'library')
Let's go to the library

>>> age = 12
>>> s = 'minor' if age < 21 else 'adult'
>>> s
'minor'

>>> 'yes' if ('qux' in ['foo', 'bar', 'baz']) else 'no'
'no'
```

Note: Python's conditional expression is similar to the `<conditional_expr> ? <expr1> : <expr2>` syntax used by many other languages—C, Perl and Java to name a few. In fact, the `?:` operator is commonly called the ternary operator in those languages, which is probably the reason Python's conditional expression is sometimes referred to as the Python ternary operator.

You can see in PEP 308 that the `<conditional_expr> ? <expr1> : <expr2>` syntax was considered for Python but ultimately rejected in favor of the syntax shown above.

A common use of the conditional expression is to select a variable assignment. For example, suppose you want to find the larger of two numbers. Of course, there is a built-in function `max()` that does just this (and more) that you could use. But suppose you want to write your own code from scratch.

You could use a standard `if` statement with an `else` clause:

```
>>>
>>> if a > b:
...     m = a
... else:
...     m = b
...
```

But a conditional expression is shorter and arguably more readable as well:

```
>>>
>>> m = a if a > b else b
```

Remember that the conditional expression behaves like an expression syntactically. It can be used as part of a longer expression. The conditional expression has lower precedence than virtually all the other operators, so parentheses are needed to group it by itself.

In the following example, the `+` operator binds more tightly than the conditional expression, so `1 + x` and `y + 2` are evaluated first, followed by the conditional expression. The parentheses in the second case are unnecessary and do not change the result:

```
>>>
>>> x = y = 40

>>> z = 1 + x if x > y else y + 2
>>> z
42

>>> z = (1 + x) if x > y else (y + 2)
>>> z
42
```

If you want the conditional expression to be evaluated first, you need to surround it with grouping parentheses. In the next example, `(x if x > y else y)` is evaluated first. The result is `y`, which is `40`, so `z` is assigned `1 + 40 + 2` = `43`:

```
>>>
```

```
>>> x = y = 40

>>> z = 1 + (x if x > y else y) + 2
>>> z
43
```

If you are using a conditional expression as part of a larger expression, it probably is a good idea to use grouping parentheses for clarification even if they are not needed.

Conditional expressions also use short-circuit evaluation like compound logical expressions. Portions of a conditional expression are not evaluated if they don't need to be.

In the expression `<expr1> if <conditional_expr> else <expr2>`:

- If `<conditional_expr>` is true, `<expr1>` is returned and `<expr2>` is not evaluated.
- If `<conditional_expr>` is false, `<expr2>` is returned and `<expr1>` is not evaluated.

As before, you can verify this by using terms that would raise an error:

```
>>>
>>> 'foo' if True else 1/0
'foo'
>>> 1/0 if False else 'bar'
'bar'
```

In both cases, the `1/0` terms are not evaluated, so no exception is raised.

Conditional expressions can also be chained together, as a sort of alternative `if`/`elif`/`else` structure, as shown here:

```
>>>
>>> s = ('foo' if (x == 1) else
...      'bar' if (x == 2) else
...      'baz' if (x == 3) else
...      'qux' if (x == 4) else
```

```
...     'quux'
... )
>>> s
'baz'
```

It's not clear that this has any significant advantage over the corresponding `if`/`elif`/`else` statement, but it is syntactically correct Python.

The Python `pass` Statement

Occasionally, you may find that you want to write what is called a code stub: a placeholder for where you will eventually put a block of code that you haven't implemented yet.

In languages where token delimiters are used to define blocks, like the curly braces in Perl and C, empty delimiters can be used to define a code stub. For example, the following is legitimate Perl or C code:

```
# This is not Python
if (x)
{
}
```

Here, the empty curly braces define an empty block. Perl or C will evaluate the expression `x`, and then even if it is true, quietly do nothing.

Because Python uses indentation instead of delimiters, it is not possible to specify an empty block. If you introduce an `if` statement with `if <expr>:`, something has to come after it, either on the same line or indented on the following line.

Consider this script `foo.py`:

```
if True:

print('foo')
```

If you try to run `foo.py`, you'll get this:

```
C:\Users\john\Documents\Python\doc>python foo.py
  File "foo.py", line 3
    print('foo')
        ^
IndentationError: expected an indented block
```

The Python `pass` statement solves this problem. It doesn't change program behavior at all. It is used as a placeholder to keep the interpreter happy in any situation where a statement is syntactically required, but you don't really want to do anything:

```
if True:
    pass

print('foo')
```

Now `foo.py` runs without error:

```
C:\Users\john\Documents\Python\doc>python foo.py
foo
```

Conclusion

With the completion of this tutorial, you are beginning to write Python code that goes beyond simple sequential execution:

- You were introduced to the concept of control structures. These are compound statements that alter program control flow—the order of execution of program statements.
- You learned how to group individual statements together into a block or suite.
- You encountered your first control structure, the `if` statement, which makes it possible to conditionally execute a statement or block based on the evaluation of program data.

All of these concepts are crucial to developing more complex Python code.

The next two tutorials will present two new control structures: the `while` statement and the `for` statement. These structures facilitate iteration, execution of a statement or block of statements repeatedly.