

# Python Bootcamp Session #2

The DA Programming Club

# Conditionals: If Statement

## Python

```
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.)

```
>>> 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')
```

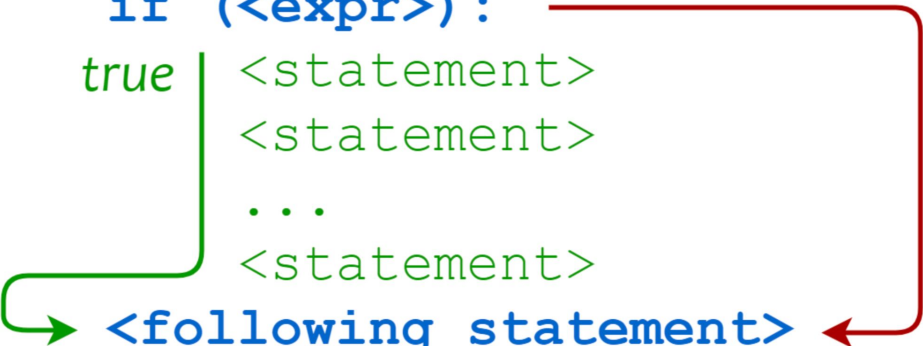
```
...
```

# Conditionals: Compound If Statement

`if (<expr>) :` *false*

*true* `<statement>`  
`<statement>`  
`...`  
`<statement>`

`<following_statement>`



```
main.py x
1 def main():
2     # Creating an empty Dictionary
3     if 'foo' in ['bar', 'baz', 'qux']:
4         print('Expression was true')
5         print('Executing code in suite')
6         print('...')
7         print('Complete!')
8         print('After conditional')
9
10 if __name__ == "__main__":
11     main()
```


Console Shell

After conditional  
>

# Conditionals: If Else

```
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.

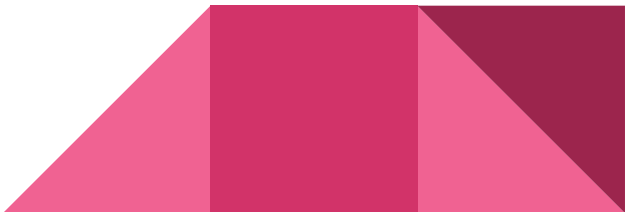


# Conditionals: Elif

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.

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:

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



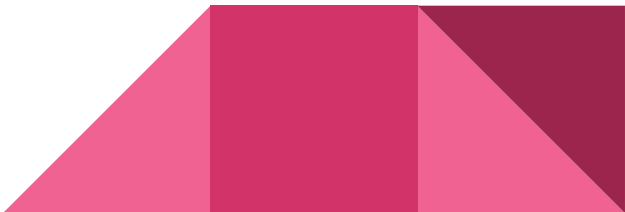
# Conditionals: One-Line If Statements

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

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

If `<expr>` is true, execute all of `<statement_1> ... <statement_n>`.

Otherwise, don't execute any of them. The `<statements>` are treated as a suite, and either all of them are executed, or none of them are.



# Conditional Expressions

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.)

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

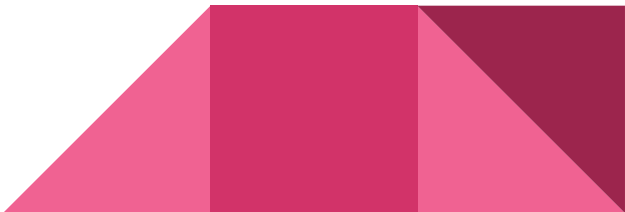
This is different from the `if` statement forms listed above because it is not a control structure that directs the flow of program execution. It acts more like an operator that defines an expression. In the above example, `<conditional_expr>` is evaluated first. If it is true, the expression evaluates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

```
>>> 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'
```



# Loops: While

```
while <expr>:  
    <statement(s)>
```

`<statement(s)>` represents the block to be repeatedly executed, often referred to as the body of the loop. This is denoted with indentation, just as in an [if statement](#).

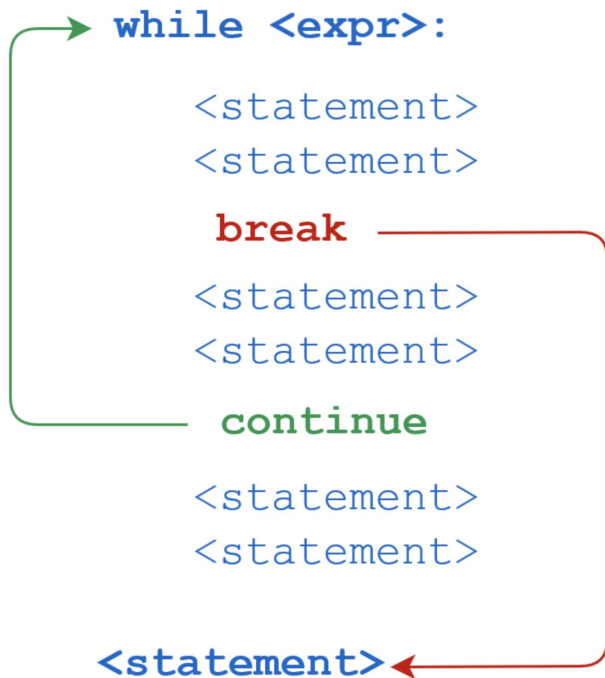
The controlling expression, `<expr>`, typically involves one or more variables that are initialized prior to starting the loop and then modified somewhere in the loop body.

When a `while` loop is encountered, `<expr>` is first evaluated in [Boolean context](#). If it is true, the loop body is executed. Then `<expr>` is checked again, and if still true, the body is executed again. This continues until `<expr>` becomes false, at which point program execution proceeds to the first statement beyond the loop body.

```
1  >>> n = 5  
2  >>> while n > 0:  
3      ...     n -= 1  
4      ...     print(n)  
5      ...  
6  4  
7  3  
8  2  
9  1  
10 0
```



# Loops: While, Break, Continue



```
1 n = 5  
2 while n > 0:  
3     n -= 1  
4     if n == 2:  
5         break  
6     print(n)  
7 print('Loop ended.')
```

Windows Console

```
C:\Users\john\Documents>python break.py  
4  
3  
Loop ended.
```

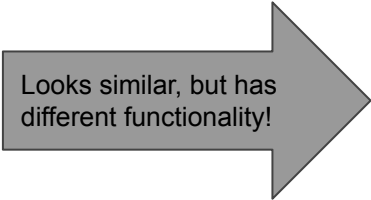
```
1 n = 5  
2 while n > 0:  
3     n -= 1  
4     if n == 2:  
5         continue  
6     print(n)  
7 print('Loop ended.')
```

Windows Console

```
C:\Users\john\Documents>python continue.py  
4  
3  
1  
0  
Loop ended.
```

# Loops: While Else

```
while <expr>:  
    <statement(s)>  
else:  
    <additional_statement(s)>
```




Looks similar, but has  
different functionality!

```
while <expr>:  
    <statement(s)>  
<additional_statement(s)>
```

What's the difference?

In the latter case, without the `else` clause, `<additional_statement(s)>` will be executed after the `while` loop terminates, no matter what.

When `<additional_statement(s)>` are placed in an `else` clause, they will be executed only if the loop terminates “by exhaustion”—that is, if the loop iterates until the controlling condition becomes false. If the loop is exited by a `break` statement, the `else` clause won't be executed.



# Loops: Iterators

In Python, iterable means an object can be used in iteration. The term is used as:

- An adjective: An object may be described as iterable.
- A noun: An object may be characterized as an iterable.

If an object is iterable, it can be passed to the built-in Python function `iter()`, which returns something called an iterator. Yes, the terminology gets a bit repetitive. Hang in there. It all works out in the end.

```
>>> iter('foobar')                                     # String
<str_iterator object at 0x036E2750>

>>> iter(['foo', 'bar', 'baz'])                         # List
<list_iterator object at 0x036E27D0>

>>> iter(('foo', 'bar', 'baz'))                         # Tuple
<tuple_iterator object at 0x036E27F0>

>>> iter({'foo', 'bar', 'baz'})                         # Set
<set_iterator object at 0x036DEA08>

>>> iter({'foo': 1, 'bar': 2, 'baz': 3})                # Dict
<dict_keyiterator object at 0x036DD990>
```

# Loops: Iterators (Cont)

An iterator is essentially a value producer that yields successive values from its associated iterable object. The built-in function `next()` is used to obtain the next value from an iterator.

```
>>> a = ['foo', 'bar', 'baz']

>>> itr = iter(a)
>>> itr
<list_iterator object at 0x031EFD10>

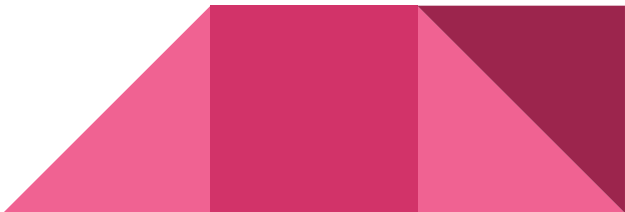
>>> next(itr)
'foo'
>>> next(itr)
'bar'
>>> next(itr)
'baz'
```

If all the values from an iterator have been returned already, a subsequent `next()` call raises a `StopIteration` exception. Any further attempts to obtain values from the iterator will fail.

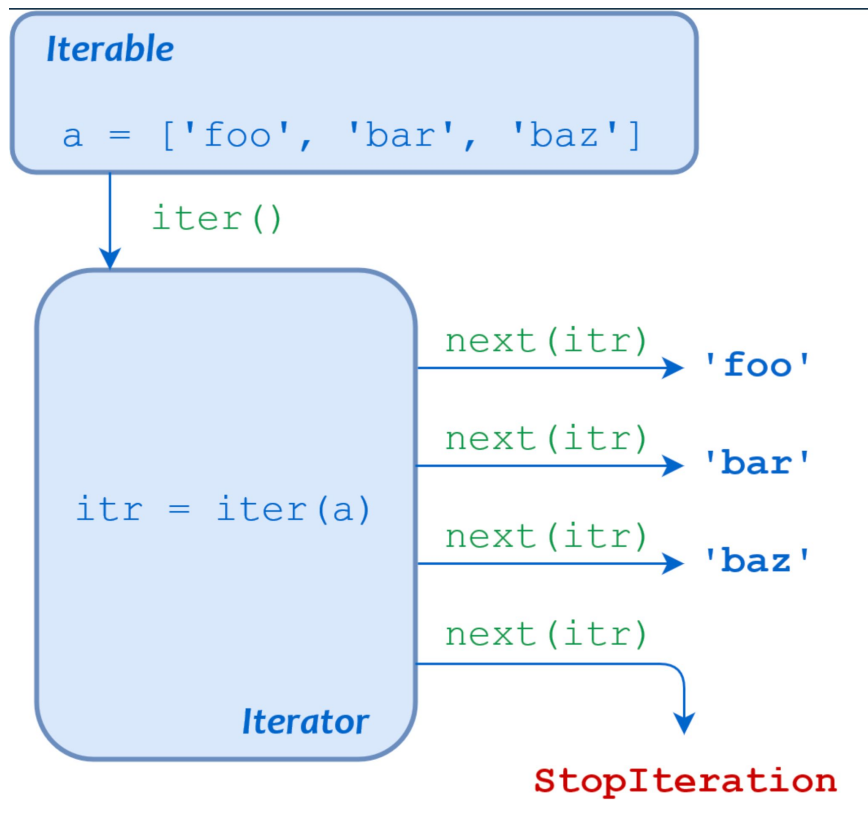
```
>>> next(itr)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    next(itr)
StopIteration
```

# Loops: Iterator (Cont)

Term	Meaning
<b>Iteration</b>	The process of looping through the objects or items in a collection
<b>Iterable</b>	An object (or the adjective used to describe an object) that can be iterated over
<b>Iterator</b>	The object that produces successive items or values from its associated iterable
<b>iter()</b>	The built-in function used to obtain an iterator from an iterable



# Loops: For Loop



```
>>> a = ['foo', 'bar', 'baz']
>>> for i in a:
...     print(i)
...
foo
bar
baz
```

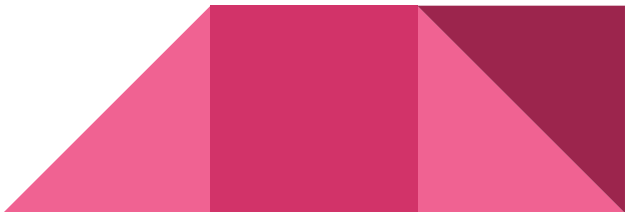
# Loops: Continue and Break

`break` terminates the loop completely and proceeds to the first statement following the loop.

`continue` terminates the current iteration and proceeds to the next iteration.

```
>>> for i in ['foo', 'bar', 'baz', 'qux']:
...     if 'b' in i:
...         break
...     print(i)
...
foo
```

```
>>> for i in ['foo', 'bar', 'baz', 'qux']:
...     if 'b' in i:
...         continue
...     print(i)
...
foo
qux
```



# Loops: For Else

The `else` clause will be executed if the loop terminates through exhaustion of the iterable:

```
>>> for i in ['foo', 'bar', 'baz', 'qux']:
...     print(i)
... else:
...     print('Done.') # Will execute
...
foo
bar
baz
qux
Done.
```





# Functions

The usual syntax for defining a Python function is as follows:

```
Python

def <function_name>([<parameters>]):
    <statement(s)>
```

The components of the definition are explained in the table below:

Component	Meaning
def	The keyword that informs Python that a function is being defined
<function_name>	A valid Python identifier that names the function
<parameters>	An optional, comma-separated list of parameters that may be passed to the function
:	Punctuation that denotes the end of the Python function header (the name and parameter list)
<statement(s)>	A block of valid Python statements

The final item, `<statement(s)>`, is called the body of the function. The body is a block of statements that will be executed when the function is called. The body of a Python function is defined by indentation in accordance with the [off-side rule](#). This is the same as code blocks associated with a control structure, like an `if` or `while` statement.




# Functions

The syntax for calling a Python function is as follows:

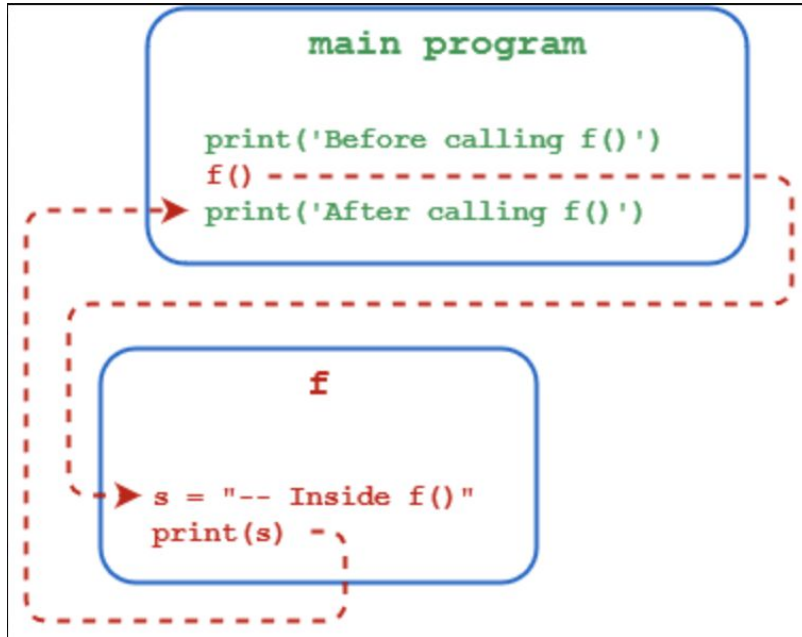
Python

```
<function_name>([<arguments>])
```

`<arguments>` are the values passed into the function. They correspond to the `<parameters>` in the Python function definition. You can define a function that doesn't take any arguments, but the parentheses are still required. Both a function definition and a function call must always include parentheses, even if they're empty.



# Functions -> Control Flow



```
1 def f():
2     s = '-- Inside f()'
3     print(s)
4
5 print('Before calling f()')
6 f()
7 print('After calling f()')
```

# Positional VS Keyword Arguments

Python

```
>>> def f(qty, item, price):  
...     print(f'{qty} {item} cost ${price:.2f}')  
... 
```

Positional

```
>>> f('bananas', 1.74, 6)  
bananas 1.74 cost $6.00
```

Keyword

```
>>> f(qty=6, item='bananas', price=1.74)  
6 bananas cost $1.74
```

Like with positional arguments, though, the number of arguments and parameters must still match. So, keyword arguments allow flexibility in the order that function arguments are specified, but the number of arguments is still rigid.

# Default Parameters

Python

```
>>> def f(qty=6, item='bananas', price=1.74):  
...     print(f'{qty} {item} cost ${price:.2f}')  
... 
```

When this version of `f()` is called, any argument that's left out assumes its default value:

```
>>> f(4, 'apples', 2.24)  
4 apples cost $2.24  
>>> f(4, 'apples')  
4 apples cost $1.74  
  
>>> f(4)  
4 bananas cost $1.74  
>>> f()  
6 bananas cost $1.74  
  
>>> f(item='kumquats', qty=9)  
9 kumquats cost $1.74  
>>> f(price=2.29)  
6 bananas cost $2.29
```

## In summary...

- **Positional arguments** must agree in order and number with the parameters declared in the function definition.
- **Keyword arguments** must agree with declared parameters in number, but they may be specified in arbitrary order.
- **Default parameters** allow some arguments to be omitted when the function is called.



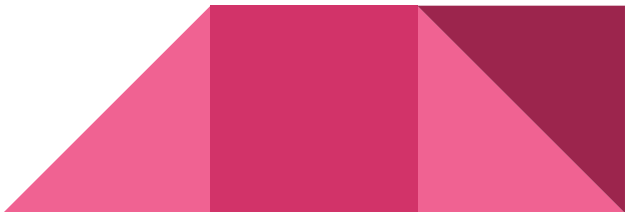
# Mutable Default Parameters

Things can get weird if you specify a default parameter value that is a mutable object. Consider this Python function definition:

```
>>> def f(my_list=[]):  
...     my_list.append('###')  
...     return my_list  
...
```

`f()` takes a single list parameter, appends the [string](#) `'###'` to the end of the list, and returns the result:

```
>>> f(['foo', 'bar', 'baz'])  
['foo', 'bar', 'baz', '###']  
  
>>> f([1, 2, 3, 4, 5])  
[1, 2, 3, 4, 5, '###']
```



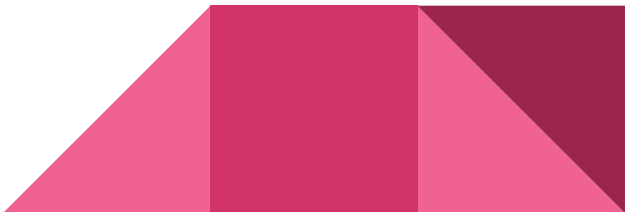
# Mutable Default Parameters (Cont)

The default value for parameter `my_list` is the empty list, so if `f()` is called without any arguments, then the return value is a list with the single element `'###'`:

```
>>> f()  
['###']
```

Everything makes sense so far. Now, what would you expect to happen if `f()` is called without any parameters a second and a third time? Let's see:

```
>>> f()  
['###', '###']  
>>> f()  
['###', '###', '###']
```





# Mutable Default Parameters (Cont)

In Python, default parameter values are defined only once when the function is defined (that is, when the `def` statement is executed). The default value isn't re-defined each time the function is called. Thus, each time you call `f()` without a parameter, you're performing `.append()` on the same list. You can demonstrate this with `id()`.

The object identifier displayed confirms that, when `my_list` is allowed to default, the value is the same object with each call. Since lists are mutable, each subsequent `.append()` call causes the list to get longer. This is a common and pretty well-documented **pitfall when you're using a mutable object as a parameter's default value**. It potentially leads to confusing code behavior, and is probably best avoided.

```
>>> def f(my_list=[]):
...     print(id(my_list))
...     my_list.append('###')
...     return my_list
...
>>> f()
140095566958408
['###']
>>> f()
140095566958408
['###', '###']
>>> f()
140095566958408
['###', '###', '###']
```

# Mutable Default Parameters (Cont)

As a workaround, consider using a default argument value that signals no argument has been specified. Most any value would work, but `None` is a common choice. When the sentinel value indicates no argument is given, create a new empty list inside the function.

Note how this ensures that `my_list` now truly defaults to an empty list whenever `f()` is called without an argument.

```
>>> def f(my_list=None):
...     if my_list is None:
...         my_list = []
...     my_list.append('###')
...     return my_list
...

>>> f()
['###']

>>> f()
['###']

>>> f()
['###']

>>> f(['foo', 'bar', 'baz'])
['foo', 'bar', 'baz', '###']

>>> f([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5, '###']
```

# Resources

<https://www.learnpython.org>

<https://pythonbasics.org>

<https://realpython.com>

[https://www.w3schools.com/python/python\\_reference.asp](https://www.w3schools.com/python/python_reference.asp)

