STATS 507
Data Analysis in Python

Lecture 2: Functions, Conditionals, Recursion and Iteration

Functions in Python

We’ve already seen examples of functions: e.g., `type()` and `print()`

Function calls take the form `function_name(function arguments)`

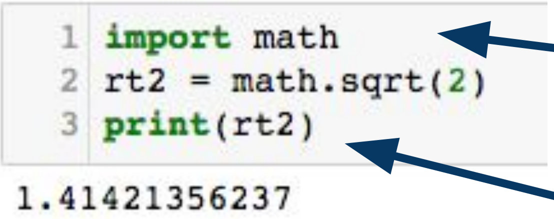
A function takes zero or more **arguments** and **returns** a value

Functions in Python

We’ve already seen examples of functions: e.g., `type()` and `print()`

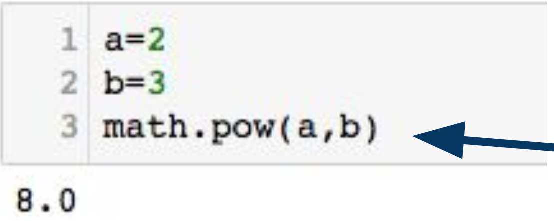
Function calls take the form `function_name(function arguments)`

A function takes zero or more **arguments** and **returns** a value



```python
import math
rt2 = math.sqrt(2)
print(rt2)
```

```
1.41421356237
```



```python
a=2
b=3
math.pow(a,b)
```

```
8.0
```

Python math **module** provides a number of math functions. We have to **import** (i.e., load) the module before we can use it.

`math.sqrt()` takes one argument, returns its square root.

`math.pow()` takes two arguments. Returns the value obtained by raising the first to the power of the second.

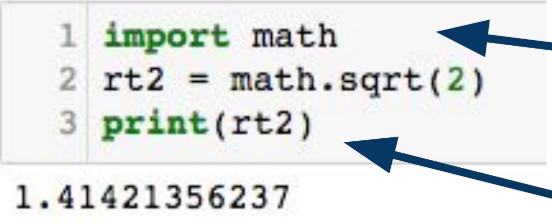
Functions in Python

Note: in the examples below, we write `math.sqrt()` to call the `sqrt()` function from the `math` module. This notation will show up a lot this semester, so get used to it!

We’ve already seen examples of functions: e.g., `type()` and `print()`

Function calls take the form `function_name(function arguments)`

A function takes zero or more **arguments** and **returns** a value

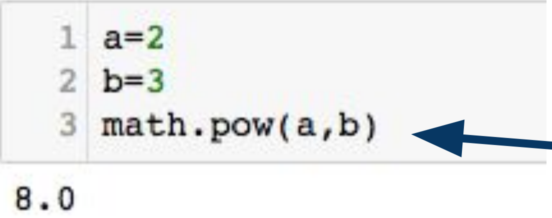


```
import math
rt2 = math.sqrt(2)
print(rt2)
```

```
1.41421356237
```

Python math **module** provides a number of math functions. We have to **import** (i.e., load) the module before we can use it.

`math.sqrt()` takes one argument, returns its square root.



```
a=2
b=3
math.pow(a,b)
```

```
8.0
```

`math.pow()` takes two arguments. Returns the value obtained by raising the first to the power of the second.

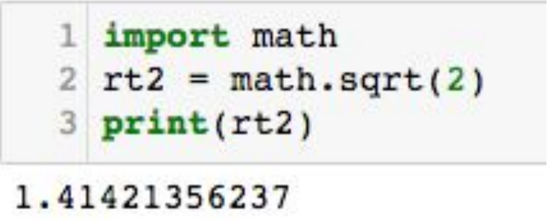
Functions in Python

Note: in the examples below, we write `math.sqrt()` to call the `sqrt()` function from the `math` module. This notation will show up a lot this semester, so get used to it!

We’ve already seen examples of functions: e.g., `type()` and `print()`

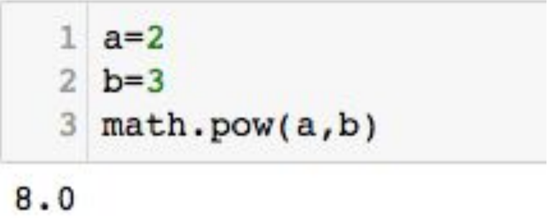
Function calls take the form `function_name(function arguments)`

A function takes zero or more **arguments** and **returns** a value



```
import math
rt2 = math.sqrt(2)
print(rt2)
```

```
1.41421356237
```



```
a=2
b=3
math.pow(a,b)
```

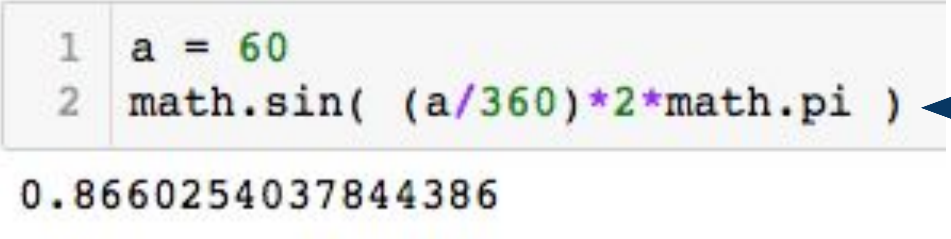
```
8.0
```

Documentation for the Python `math` module: https://docs.python.org/3/library/math.html

Functions in Python

Functions can be **composed**

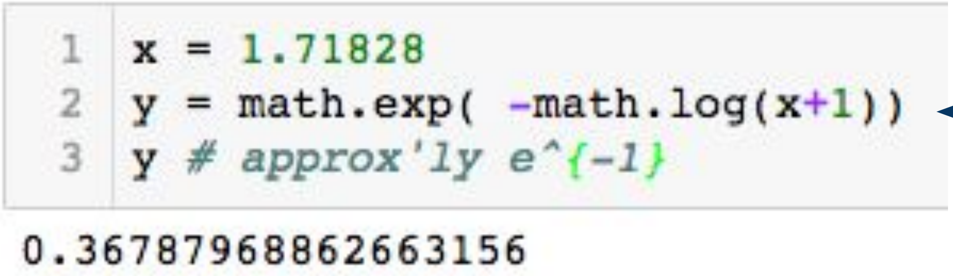
- Supply an expression as the argument of a function
- Output of one function becomes input to another



```
a = 60
math.sin( (a/360)*2*math.pi )
```

```
0.8660254037844386
```

`math.sin()` has as its argument an expression, which has to be evaluated before we can compute the answer.



```
x = 1.71828
y = math.exp( -math.log(x+1))
y # approx'ly e^{-1}
```

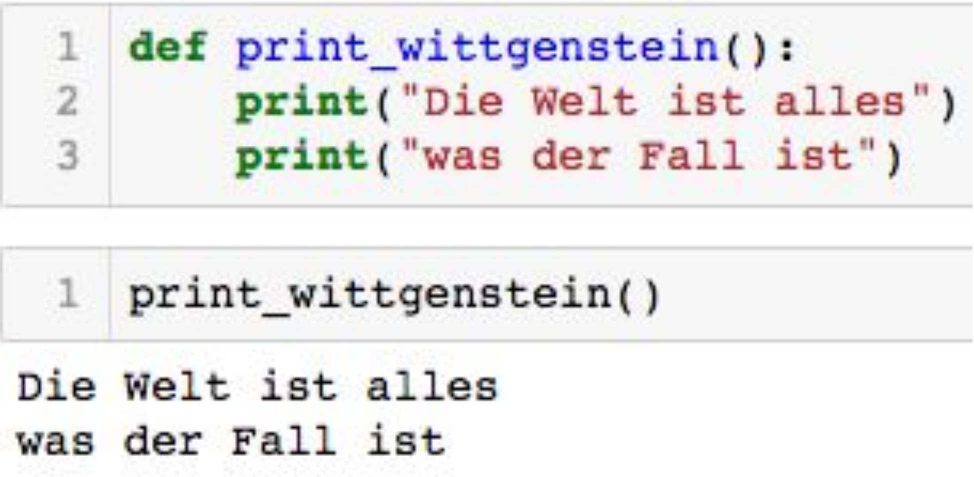
```
0.36787968862663156
```

Functions can even have the outputs of other functions as their arguments.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```
print_wittgenstein()
```

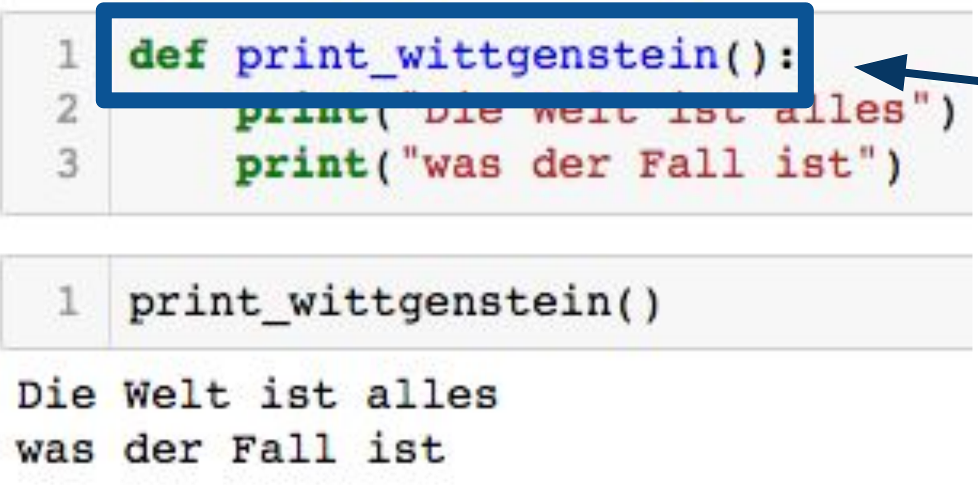
```
Die Welt ist alles
was der Fall ist
```

Let’s walk through this line by line.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```
print_wittgenstein()
```

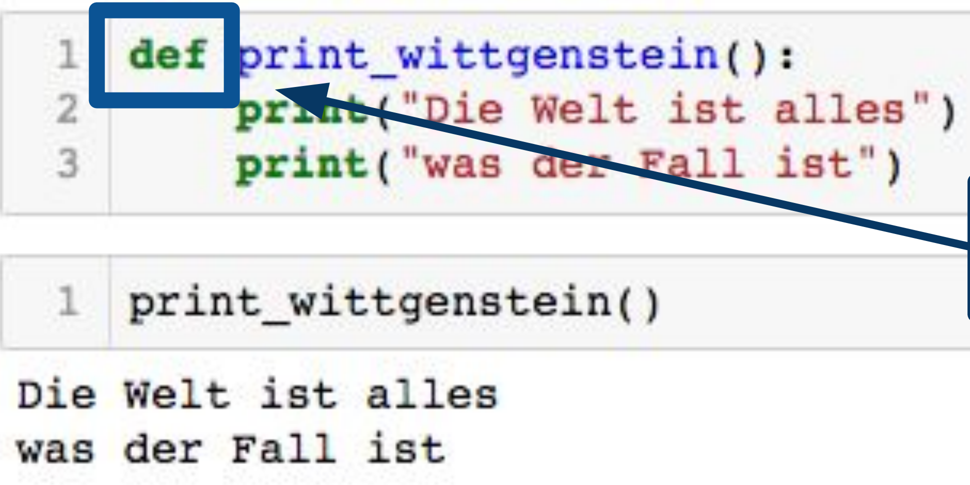
```
Die Welt ist alles
was der Fall ist
```

This line (called the **header** in some documentation) says that we are defining a function called `print_wittgenstein`, and that the function takes no argument.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```
print_wittgenstein()
```

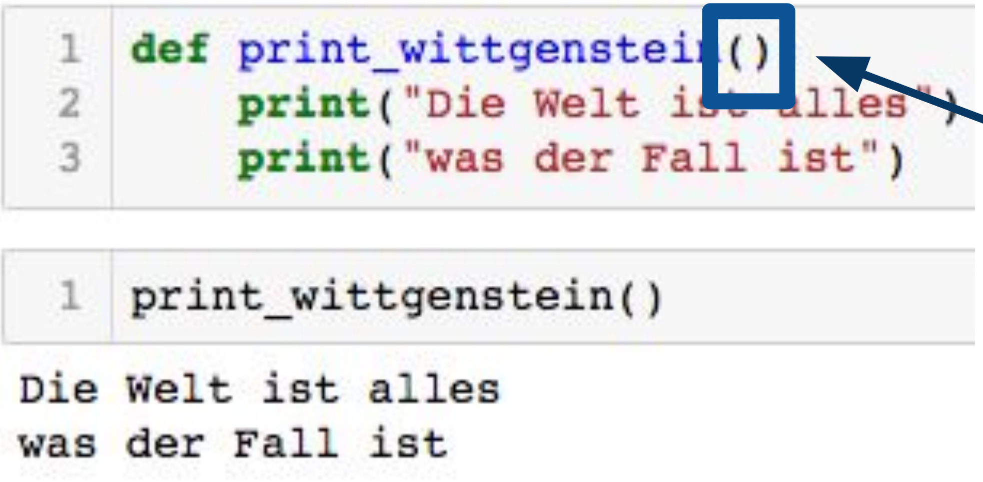
```
Die Welt ist alles
was der Fall ist
```

The `def` keyword tells Python that we are defining a function.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```
print_wittgenstein()
```

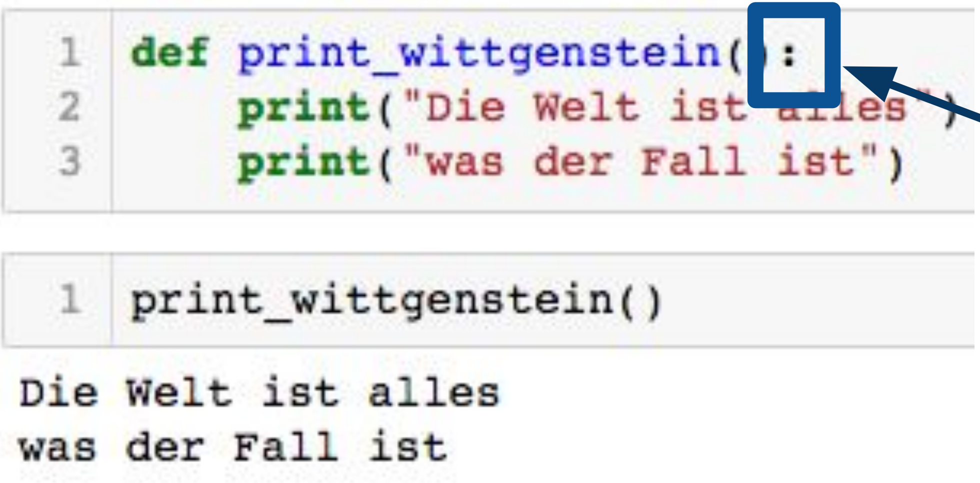
```
Die Welt ist alles
was der Fall ist
```

Any arguments to the function are giving inside the parentheses. This function takes no arguments, so we just give empty parentheses. In a few slides, we’ll see a function that takes arguments.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```
print_wittgenstein()
```

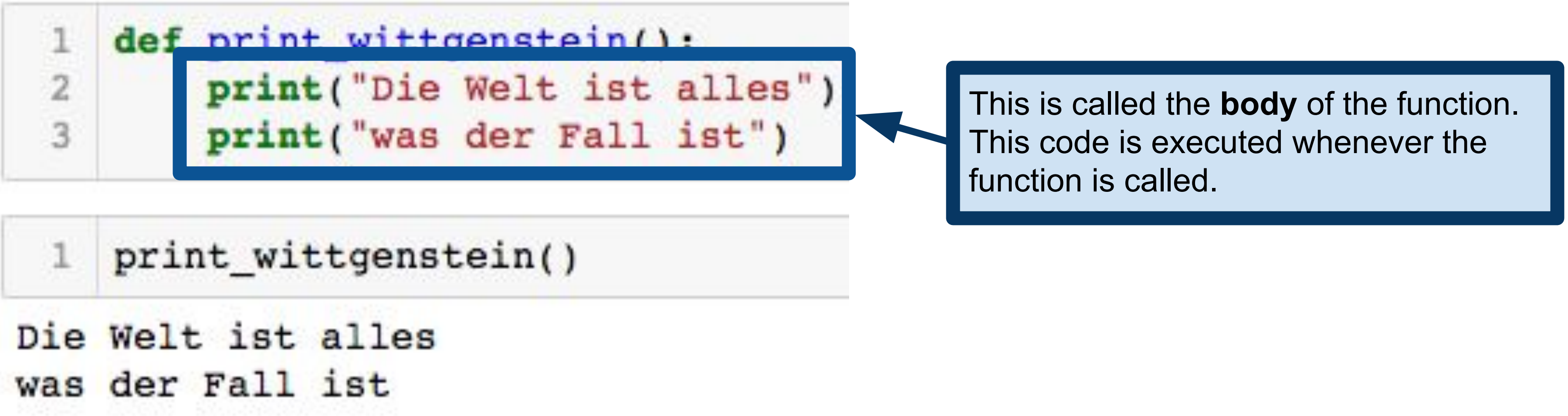
```
Die Welt ist alles
was der Fall ist
```

The colon (`:`) is required by Python’s syntax. You’ll see this symbol a lot, as it is commonly used in Python to signal the start of an indented block of code. (more on this in a few slides).

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```
print_wittgenstein()
```

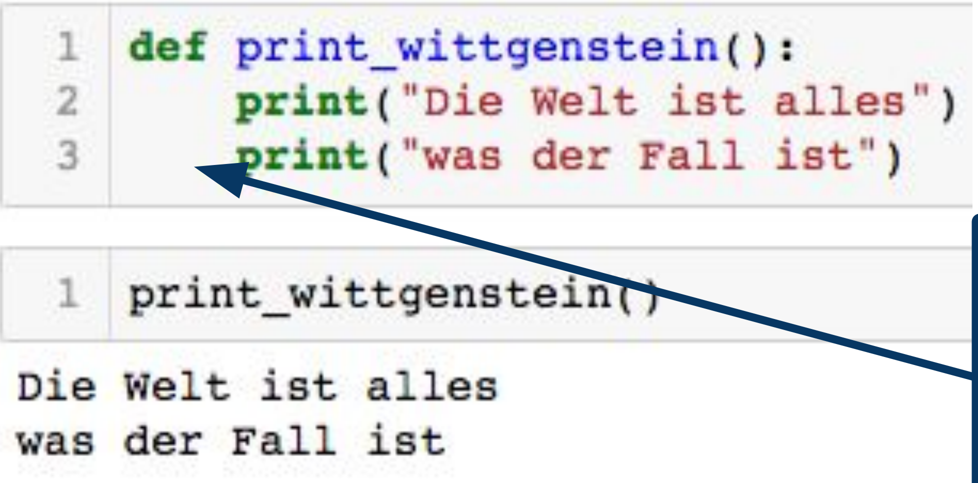
```
Die Welt ist alles
was der Fall ist
```

This is called the **body** of the function. This code is executed whenever the function is called.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```python
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```python
print_wittgenstein()
```

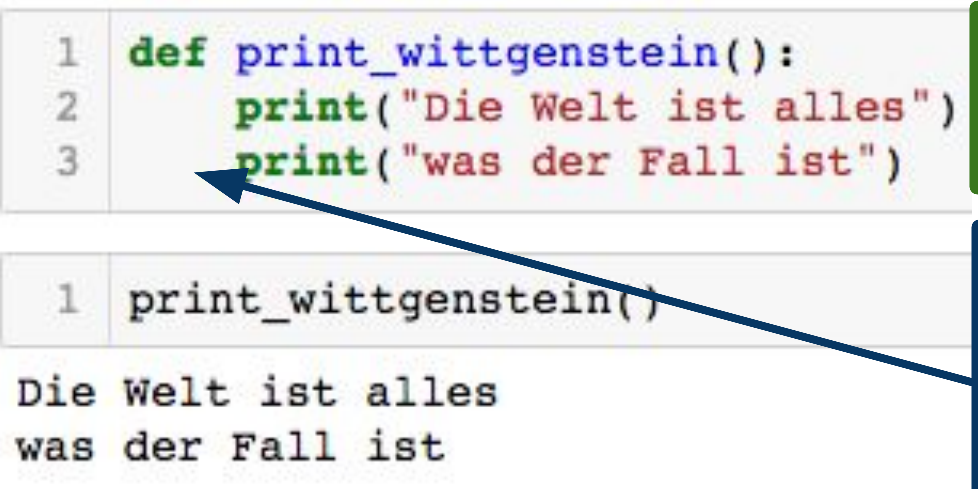
```
Die Welt ist alles
was der Fall ist
```

Note: in languages like R, C/C++ and Java, code is organized into **blocks** using curly braces ({ and }). Python is **whitespace delimited**. So we tell Python which lines of code are part of the function definition using indentation.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```
print_wittgenstein()
```

```
Die Welt ist alles
was der Fall ist
```

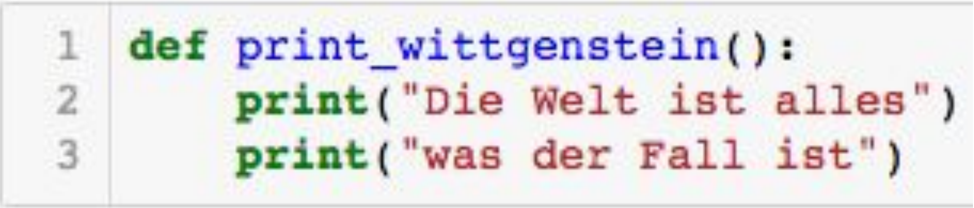
This whitespace can be tabs, or spaces, so long as it's consistent. It is taken care of automatically by most IDEs.

Note: in languages like R, C/C++ and Java, code is organized into **blocks** using curly braces (`{` and `}`). Python is **whitespace delimited**. So we tell Python which lines of code are part of the function definition using indentation.

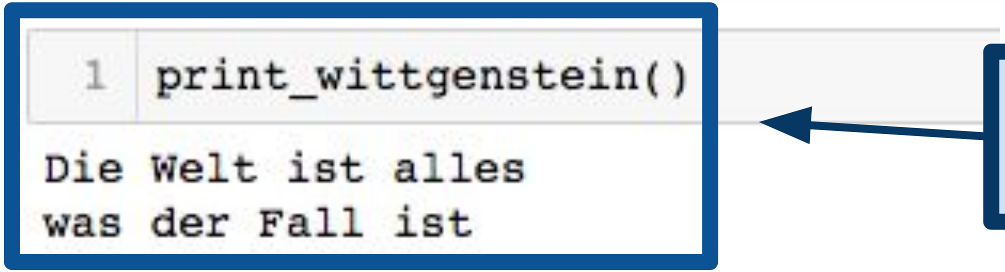
Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```python
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```



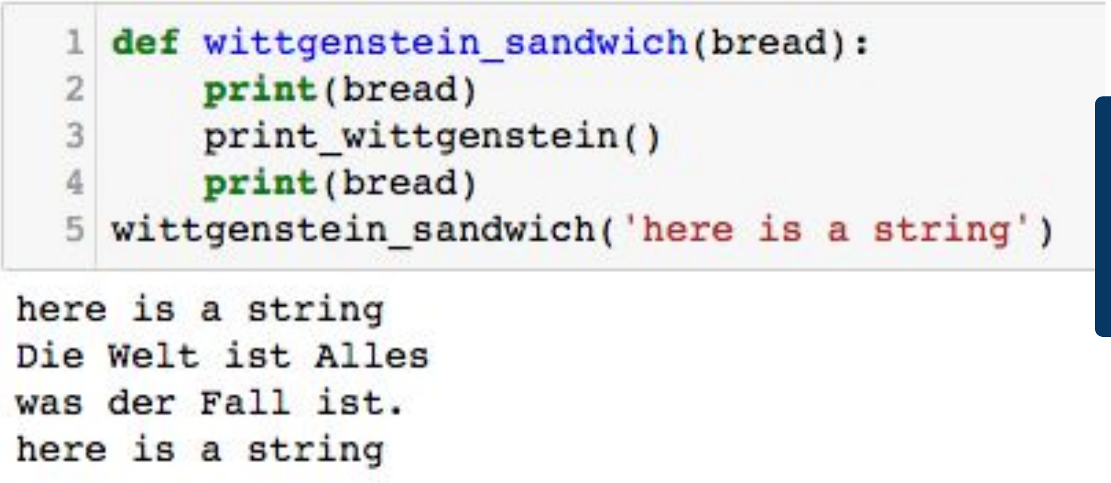
```python
print_wittgenstein()
```

```
Die Welt ist alles
was der Fall ist
```

We have defined our function. Now, any time we call it, Python executes the code in the definition, in order.

Defining Functions

After defining a function, we can use it anywhere, including in other functions



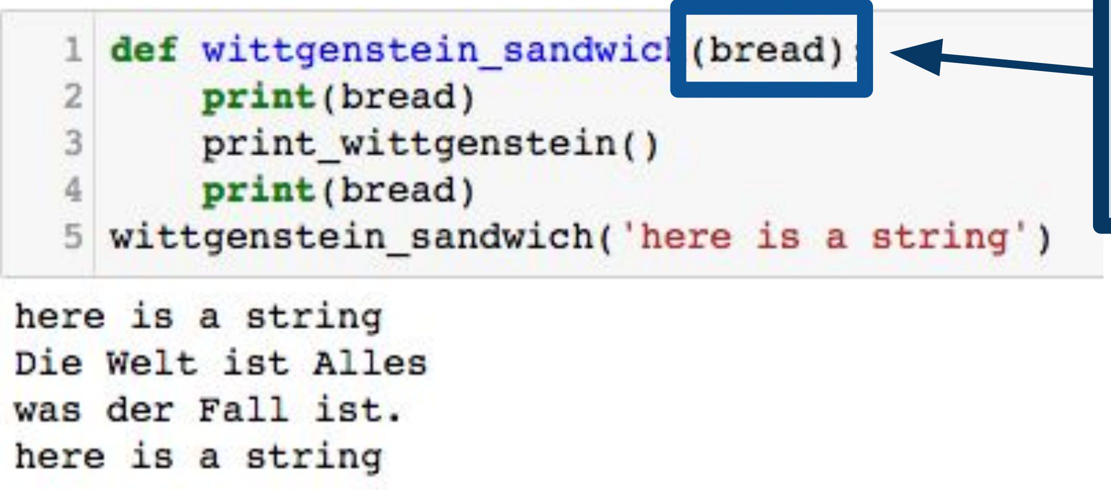
```python
def wittgenstein_sandwich(bread):
    print(bread)
    print_wittgenstein()
    print(bread)
wittgenstein_sandwich('here is a string')
```

```
here is a string
Die Welt ist Alles
was der Fall ist.
here is a string
```

This function takes one argument, prints it, then prints our Wittgenstein quote, then prints the argument again.

Defining Functions

After defining a function, we can use it anywhere, including in other functions



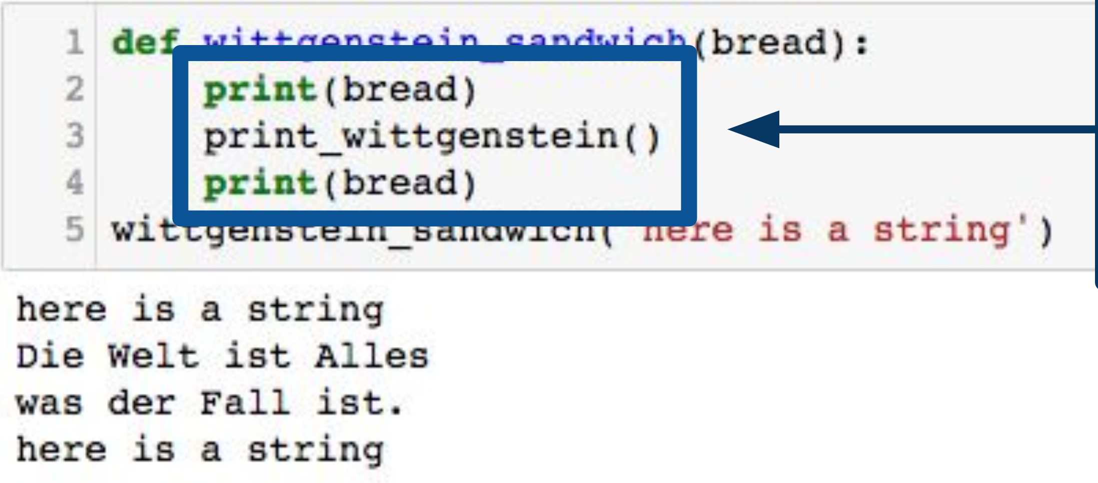
```python
def wittgenstein_sandwich(bread):
    print(bread)
    print_wittgenstein()
    print(bread)
wittgenstein_sandwich('here is a string')
```

```
here is a string
Die Welt ist Alles
was der Fall ist.
here is a string
```

This function takes one argument, which we call `bread`. All the arguments named here act like variables **within the body of the function**, but not outside the body. We'll return to this in a few slides.

Defining Functions

After defining a function, we can use it anywhere, including in other functions



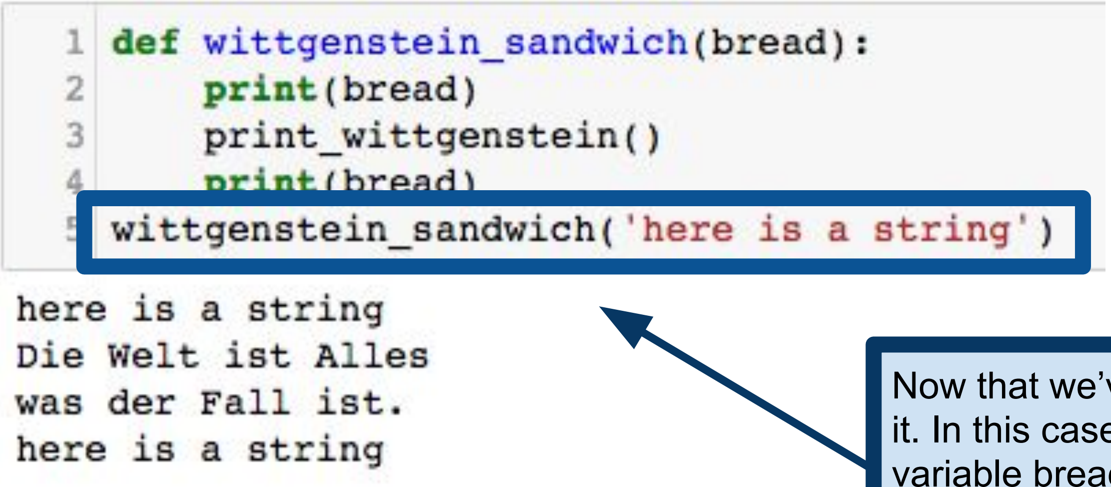
```python
def wittgenstein_sandwich(bread):
    print(bread)
    print_wittgenstein()
    print(bread)
wittgenstein_sandwich('here is a string')
```

```
here is a string
Die Welt ist Alles
was der Fall ist.
here is a string
```

Body of the function specifies what to do with the argument(s). In this case, we print whatever the argument was, then print our Wittgenstein quote, and then print the argument again.

Defining Functions

After defining a function, we can use it anywhere, including in other functions



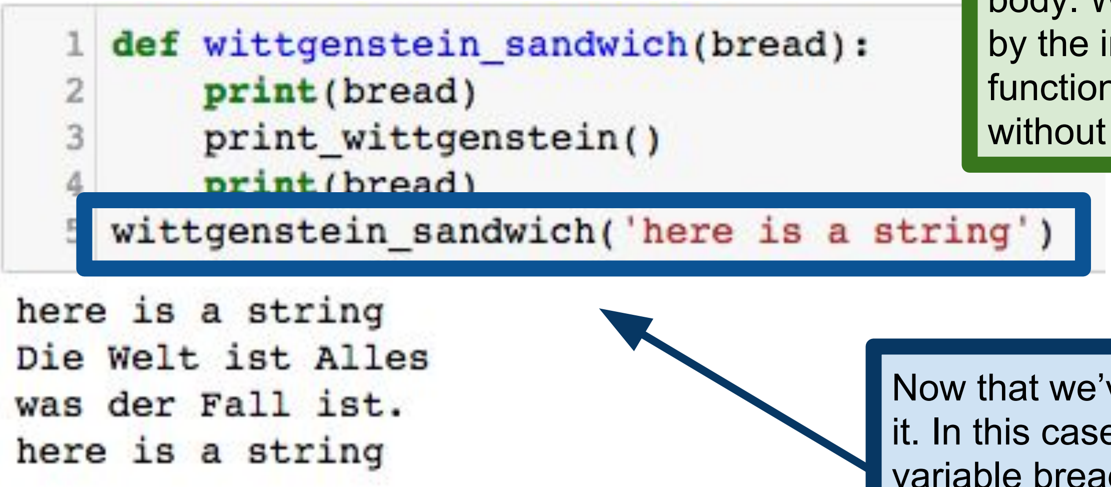
```
def wittgenstein_sandwich(bread):
    print(bread)
    print_wittgenstein()
    print(bread)
wittgenstein_sandwich('here is a string')
```

```
here is a string
Die Welt ist Alles
was der Fall ist.
here is a string
```

Now that we've defined our function, we can call it. In this case, when we call our function, the variable bread in the definition gets the value `'here is a string'` , and then proceeds to run the code in the function body.

Defining Functions

After defining a function, we can use it anywhere, including in other functions



```python
def wittgenstein_sandwich(bread):
    print(bread)
    print_wittgenstein()
    print(bread)
wittgenstein_sandwich('here is a string')
```

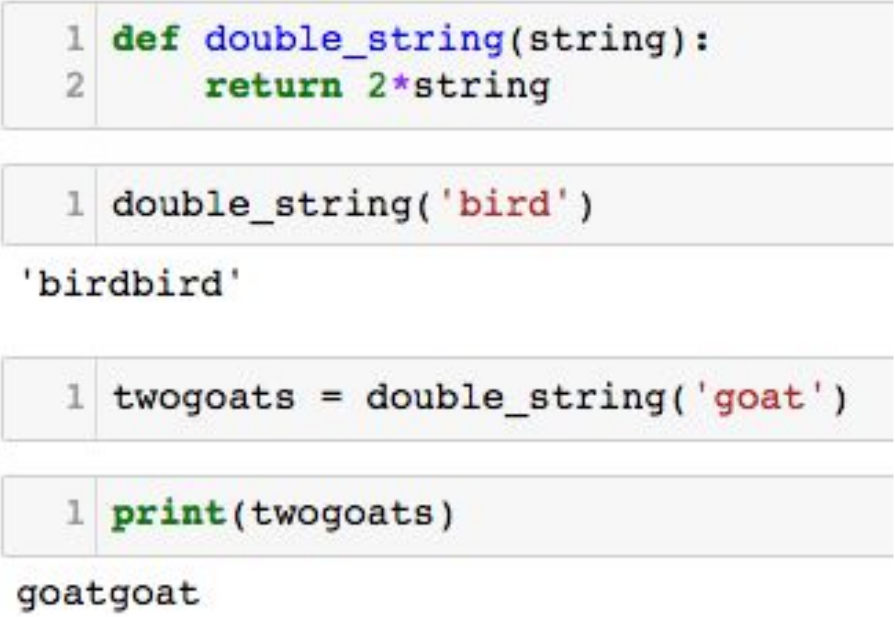
```
here is a string
Die Welt ist Alles
was der Fall ist.
here is a string
```

Note: this last line is **not** part of the function body. We communicate this fact to Python by the indentation. Python knows that the function body is finished once it sees a line without indentation.

Now that we’ve defined our function, we can call it. In this case, when we call our function, the variable bread in the definition gets the value ‘here is a string’, and then proceeds to run the code in the function body.

Defining Functions

Using the `return` keyword, we can define functions that produce results



```python
def double_string(string):
    return 2*string
```

```python
double_string('bird')
```

```
'birdbird'
```

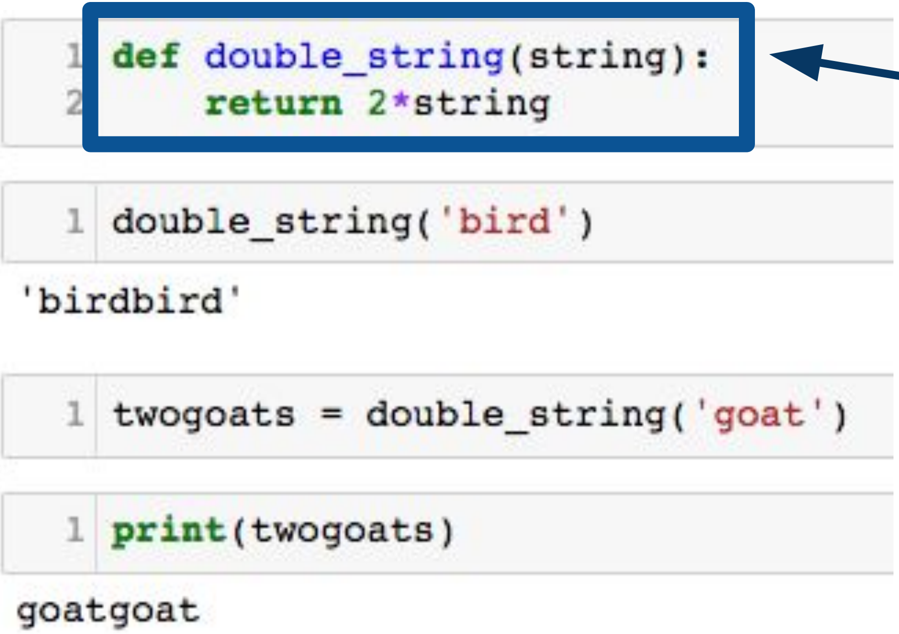
```python
twogoats = double_string('goat')
```

```python
print(twogoats)
```

```
goatgoat
```

Defining Functions

Using the `return` keyword, we can define functions that produce results



```python
def double_string(string):
    return 2*string
```

```python
double_string('bird')
```

```
'birdbird'
```

```python
twogoats = double_string('goat')
```

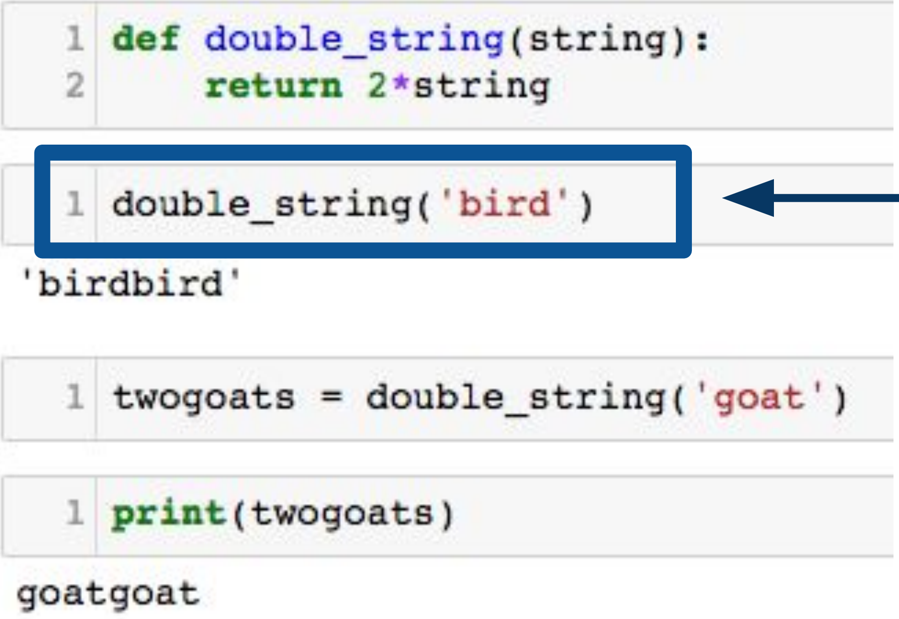
```python
print(twogoats)
```

```
goatgoat
```

`double_string` takes one argument, a string, and **returns** that string, concatenated with itself.

Defining Functions

Using the `return` keyword, we can define functions that produce results



```
def double_string(string):
    return 2*string
```

```
double_string('bird')
```

```
'birdbird'
```

```
twogoats = double_string('goat')
```

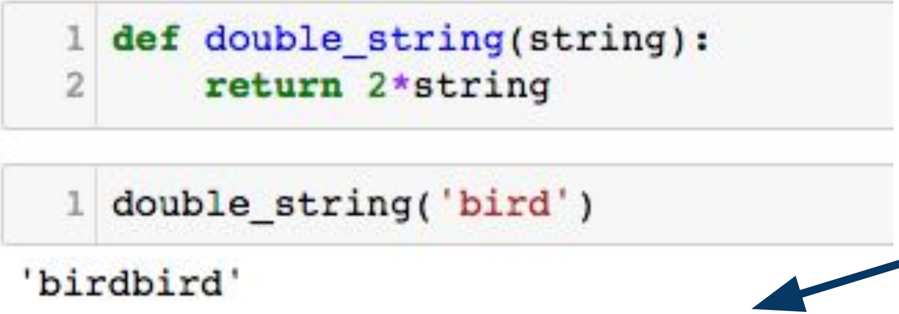
```
print(twogoats)
```

```
goatgoat
```

So when Python executes this line, it takes the string `'bird'`, which becomes the parameter `string` in the function `double_string`, and this line **evaluates** to the string `'birdbird'`.

Defining Functions

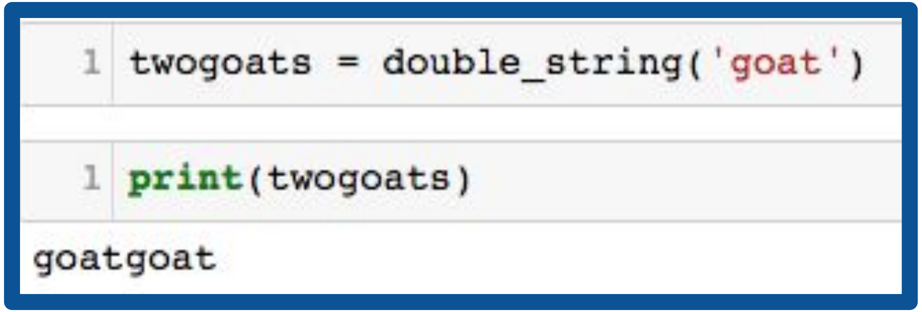
Using the `return` keyword, we can define functions that produce results



```
def double_string(string):
    return 2*string
```

```
double_string('bird')
```

```
'birdbird'
```



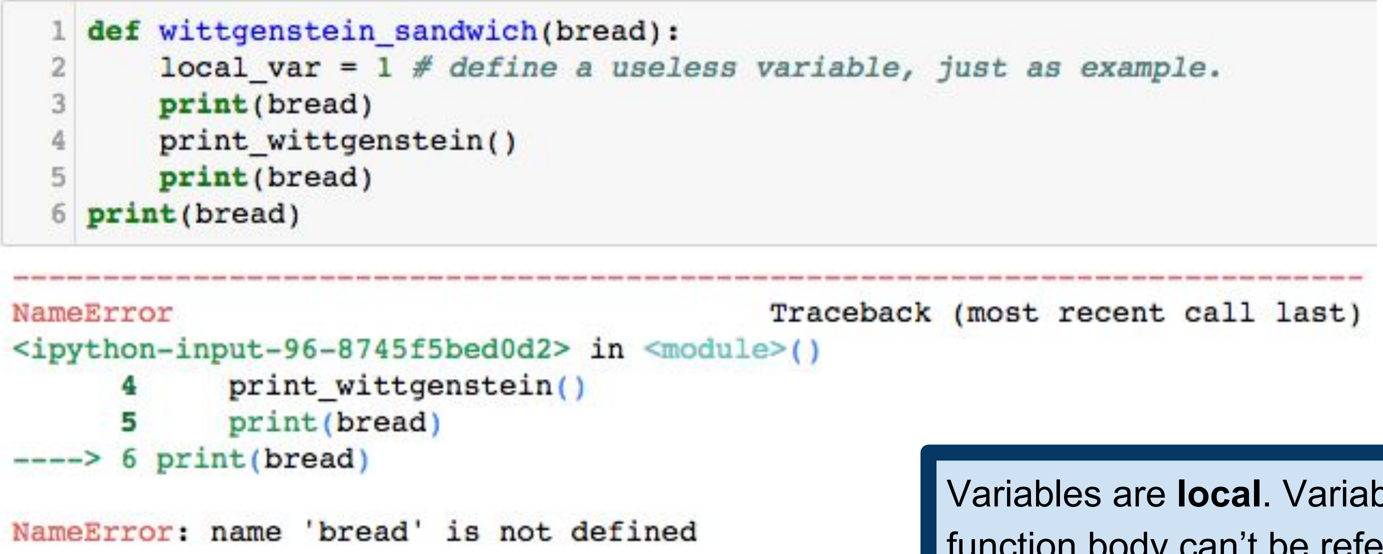
```
twogoats = double_string('goat')
```

```
print(twogoats)
```

```
goatgoat
```

Alternatively, we can call the function and assign its result to a variable, just like we did with the functions in the `math` module.

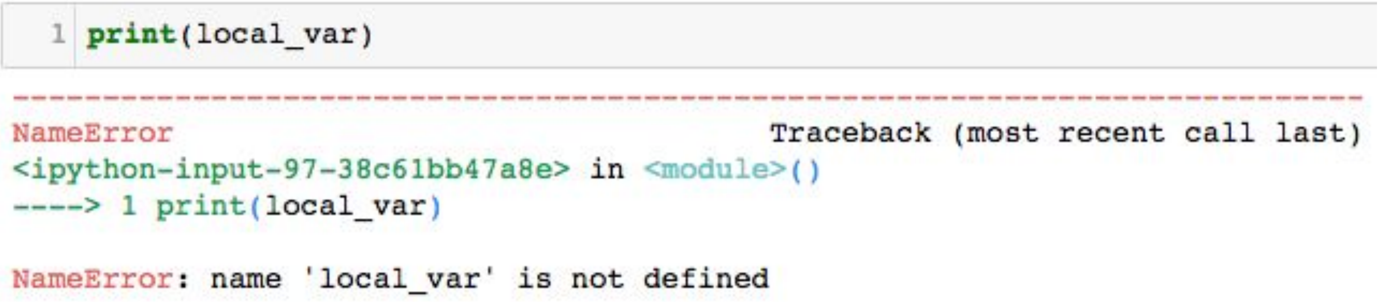
Defining Functions



```
def wittgenstein_sandwich(bread):
    local_var = 1 # define a useless variable, just as example.
    print(bread)
    print_wittgenstein()
    print(bread)
print(bread)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-96-8745f5bed0d2> in <module>()
      4     print_wittgenstein()
      5     print(bread)
----> 6 print(bread)

NameError: name 'bread' is not defined
```

Variables are **local**. Variables defined inside a function body can’t be referenced outside.



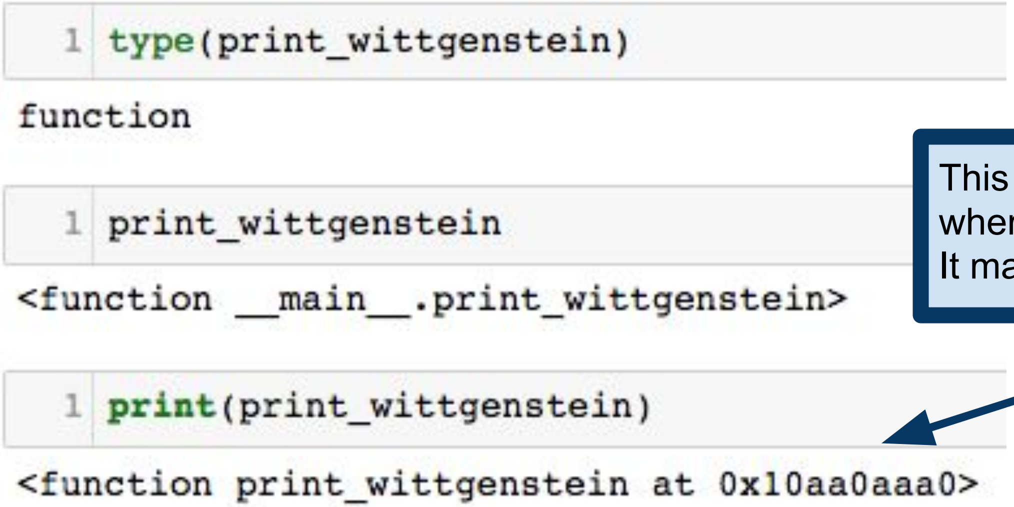
```
print(local_var)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-97-38c61bb47a8e> in <module>()
----> 1 print(local_var)

NameError: name 'local_var' is not defined
```

Defining Functions

When you define a function, you are actually creating a variable of type **function**

Functions are objects that you can treat just like other variables



```
type(print_wittgenstein)
```

```
function
```

```
print_wittgenstein
```

```
<function __main__.print_wittgenstein>
```

```
print(print_wittgenstein)
```

```
<function print_wittgenstein at 0x10aa0aaa0>
```

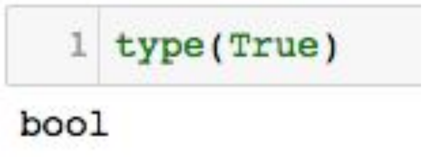
This number is the address in memory where `print_wittgenstein` is stored. It may be different on your computer.

Boolean Expressions

Boolean expressions evaluate the truth/falsity of a statement

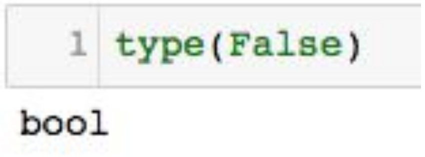
Python supplies a special Boolean type, `bool`

variable of type `bool` can be either `True` or `False`



```
type(True)
```

```
bool
```

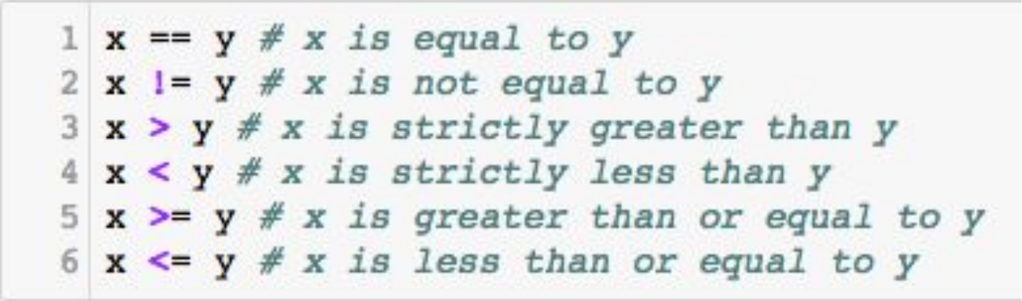


```
type(False)
```

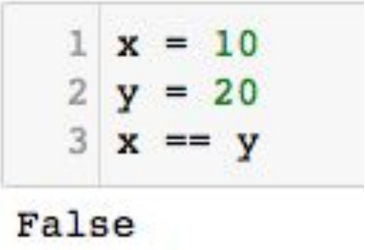
```
bool
```

Boolean Expressions

Comparison operators available in Python:

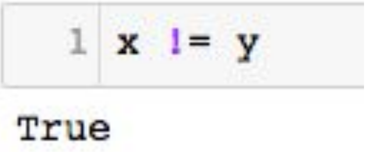


```
x == y # x is equal to y
x != y # x is not equal to y
x > y # x is strictly greater than y
x < y # x is strictly less than y
x >= y # x is greater than or equal to y
x <= y # x is less than or equal to y
```



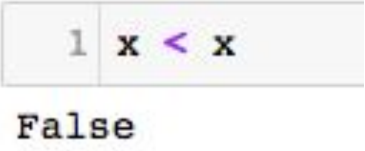
```
x = 10
y = 20
x == y
```

```
False
```



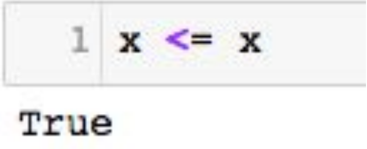
```
x != y
```

```
True
```



```
x < x
```

```
False
```



```
x <= x
```

```
True
```

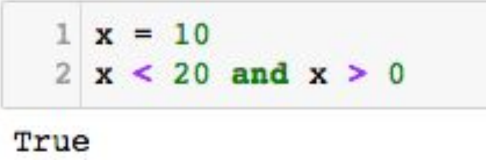
Expressions involving comparison operators evaluate to a Boolean.

Note: In true Pythonic style, one can compare many types, not just numbers. Most obviously, strings can be compared, with ordering given alphabetically.

Boolean Expressions

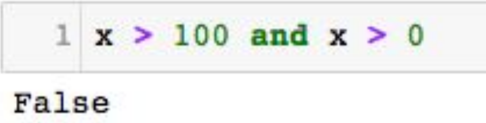
Can combine Boolean expressions into larger expressions via **logical operators**

In Python: `and`, `or` and `not`



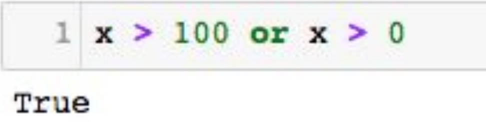
```
x = 10
x < 20 and x > 0
```

```
True
```



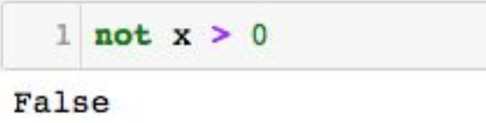
```
x > 100 and x > 0
```

```
False
```



```
x > 100 or x > 0
```

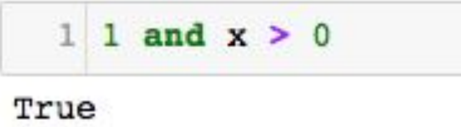
```
True
```



```
not x > 0
```

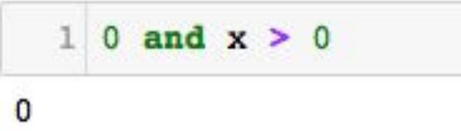
```
False
```

Note: technically, any nonzero number or any nonempty string will evaluate to `True`, but you should avoid comparing anything that isn't Boolean.



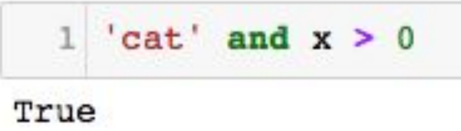
```
1 and x > 0
```

```
True
```



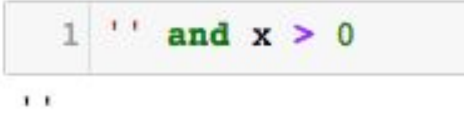
```
0 and x > 0
```

```
0
```



```
'cat' and x > 0
```

```
True
```

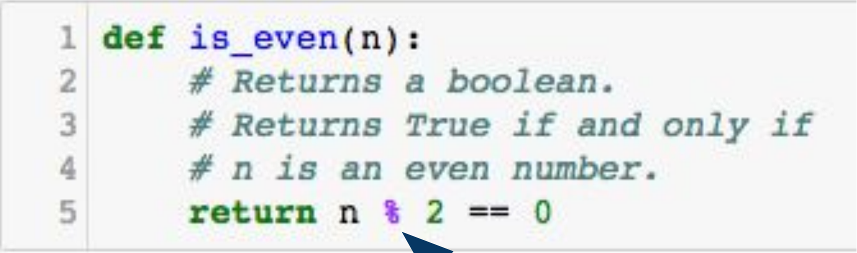


```
'' and x > 0
```

```
''
```

Boolean Expressions: Example

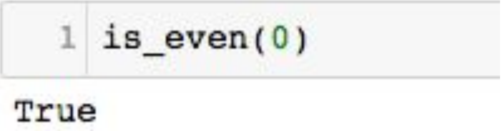
Let’s see Boolean expressions in action



```python
def is_even(n):
    # Returns a boolean.
    # Returns True if and only if
    # n is an even number.
    return n % 2 == 0
```

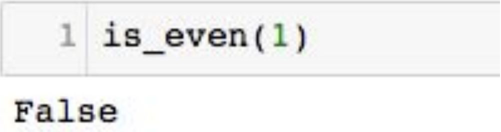
Reminder: `x % y` returns the remainder when `x` is divided by `y`.

Note: in practice, we would want to include some extra code to check that `n` is actually a number, and to “fail gracefully” if it isn’t, e.g., by throwing an error with a useful error message. More about this in future lectures.



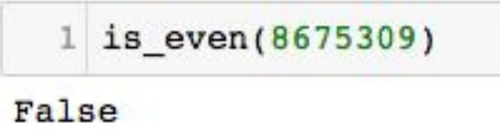
```
is_even(0)
```

```
True
```



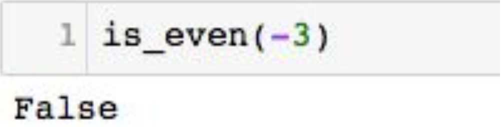
```
is_even(1)
```

```
False
```



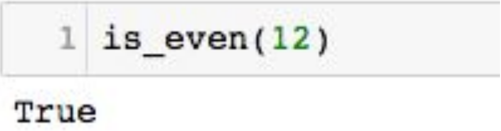
```
is_even(8675309)
```

```
False
```



```
is_even(-3)
```

```
False
```

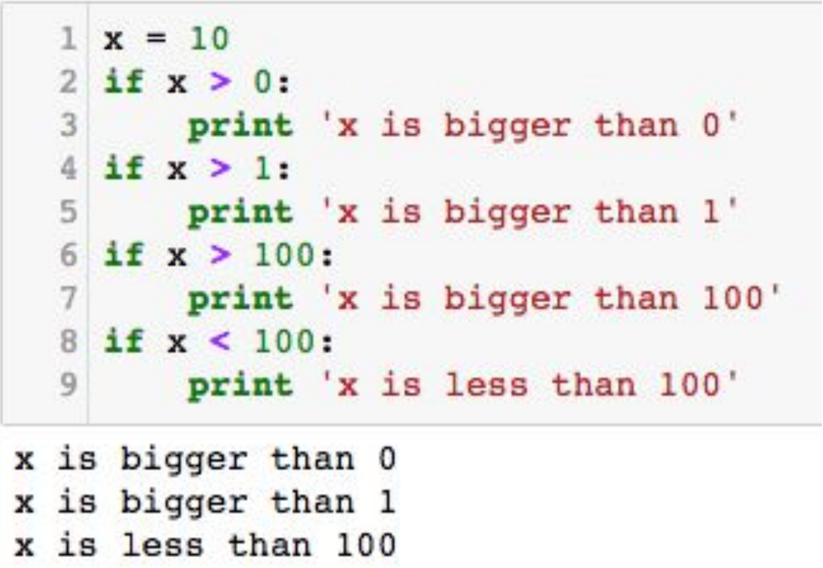


```
is_even(12)
```

```
True
```

Conditional Expressions

Sometimes we want to do different things depending on certain conditions

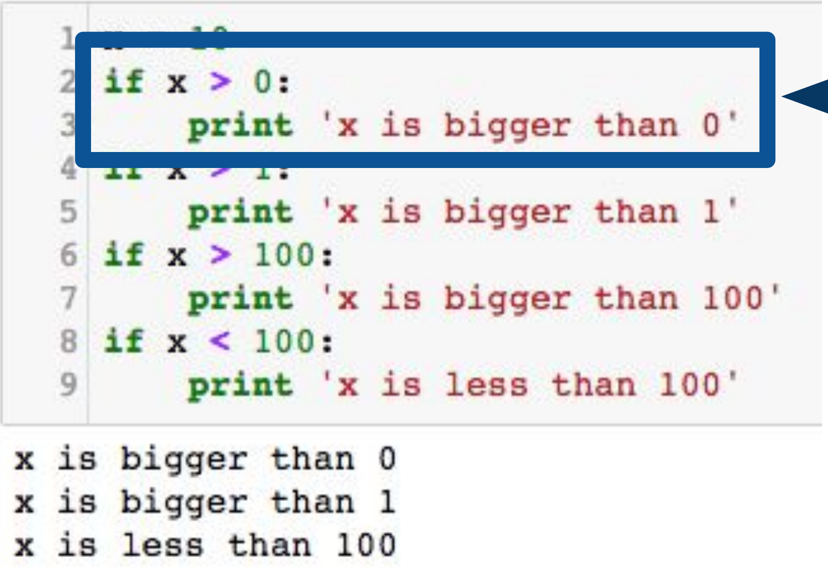


```
x = 10
if x > 0:
    print 'x is bigger than 0'
if x > 1:
    print 'x is bigger than 1'
if x > 100:
    print 'x is bigger than 100'
if x < 100:
    print 'x is less than 100'
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

Conditional Expressions

Sometimes we want to do different things depending on certain conditions



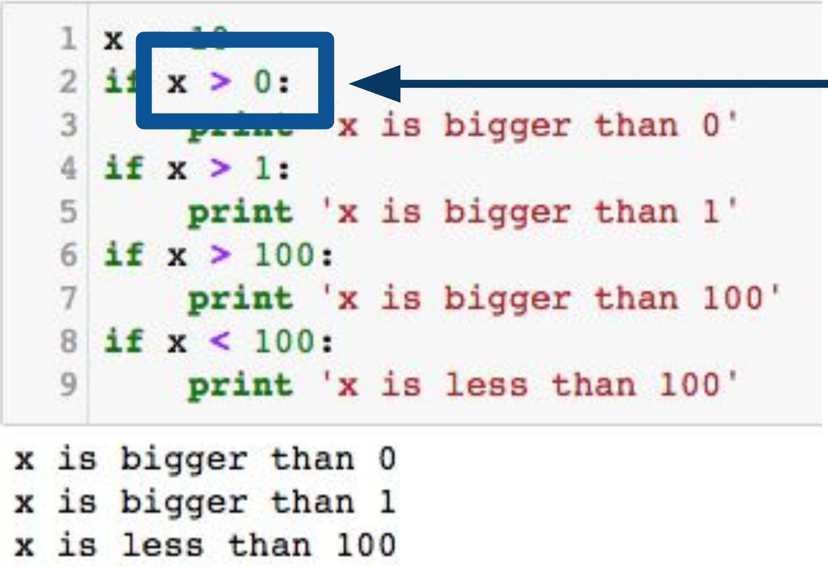
```
x = 10
if x > 0:
    print 'x is bigger than 0'
if x > 1:
    print 'x is bigger than 1'
if x > 100:
    print 'x is bigger than 100'
if x < 100:
    print 'x is less than 100'
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

This is an **if-statement**.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions



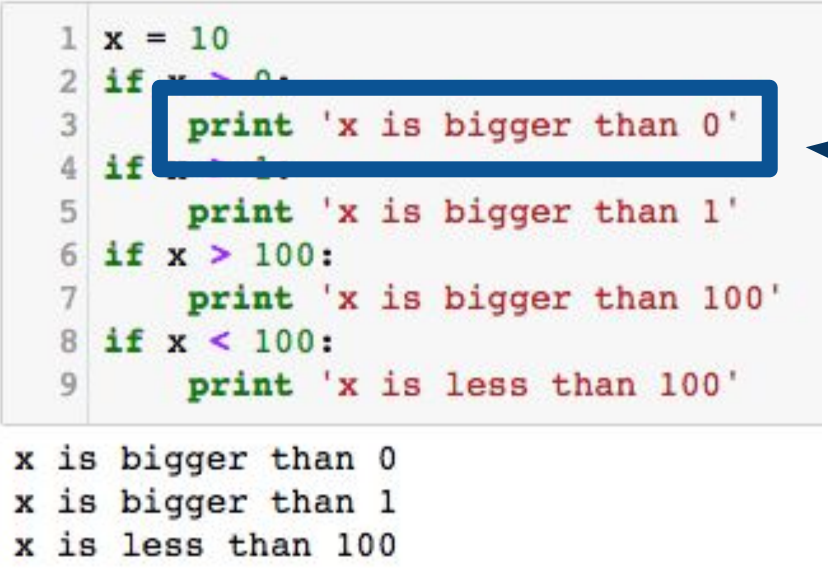
```
x = 10
if x > 0:
    print 'x is bigger than 0'
if x > 1:
    print 'x is bigger than 1'
if x > 100:
    print 'x is bigger than 100'
if x < 100:
    print 'x is less than 100'
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

This Boolean expression is called the **test condition**, or just the **condition**.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions



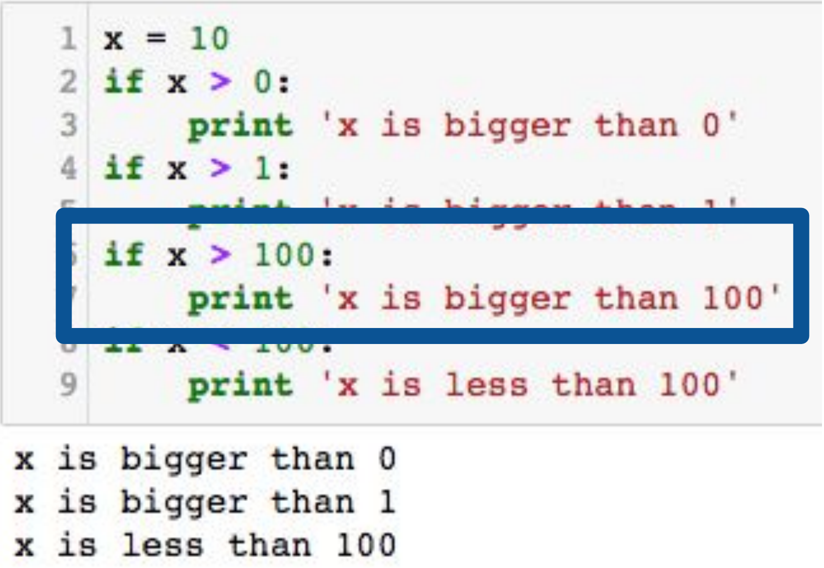
```
x = 10
if x > 0:
    print 'x is bigger than 0'
if x > 1:
    print 'x is bigger than 1'
if x > 100:
    print 'x is bigger than 100'
if x < 100:
    print 'x is less than 100'
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

If the condition evaluates to `True`, then Python runs the code in the body of the if-statement.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions



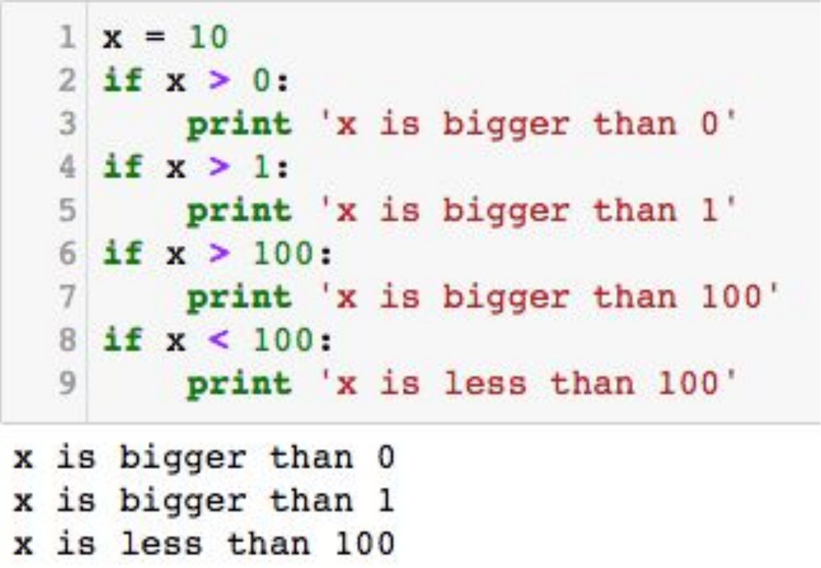
```
x = 10
if x > 0:
    print 'x is bigger than 0'
if x > 1:
    print 'x is bigger than 1'
if x > 100:
    print 'x is bigger than 100'
if x < 100:
    print 'x is less than 100'
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

If the condition evaluates to `False`, then Python skips the body and continues running code starting at the end of the if-statement.

Conditional Expressions

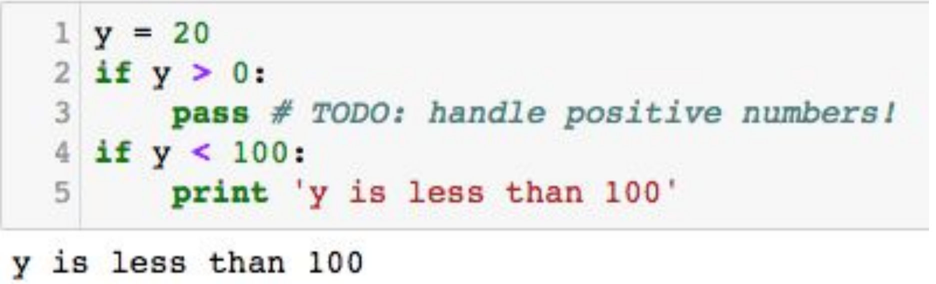
Sometimes we want to do different things depending on certain conditions



```
x = 10
if x > 0:
    print 'x is bigger than 0'
if x > 1:
    print 'x is bigger than 1'
if x > 100:
    print 'x is bigger than 100'
if x < 100:
    print 'x is less than 100'
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

Note: the body of a conditional statement can have any number of lines in it, but it must have at least one line. To do nothing, use the `pass` keyword.

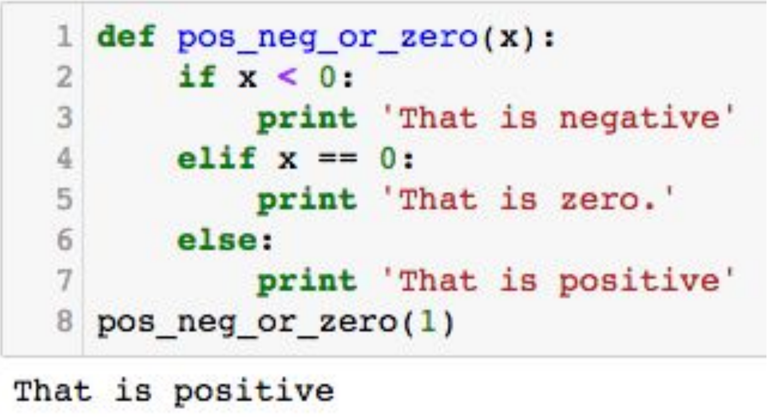


```
y = 20
if y > 0:
    pass # TODO: handle positive numbers!
if y < 100:
    print 'y is less than 100'
```

```
y is less than 100
```

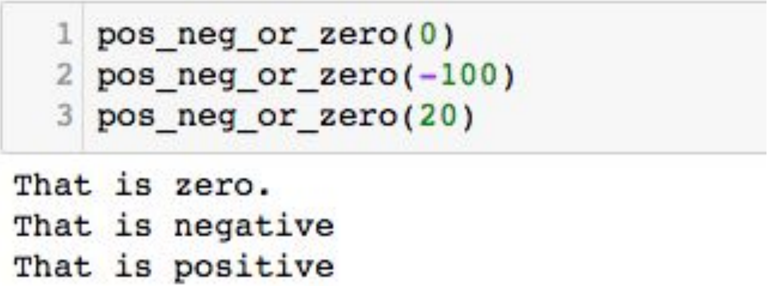
Conditional Expressions

More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

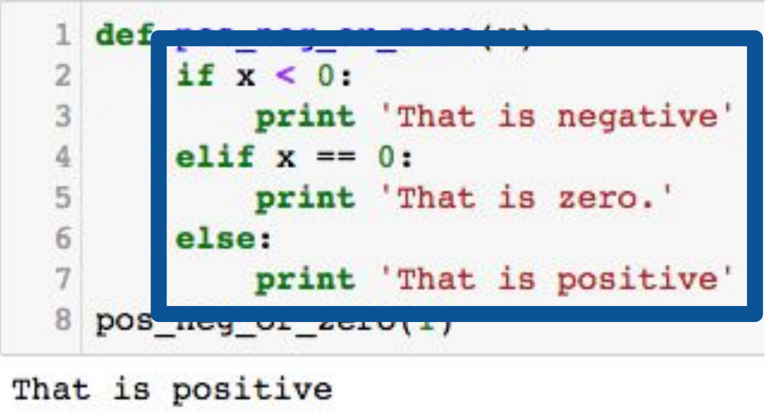


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

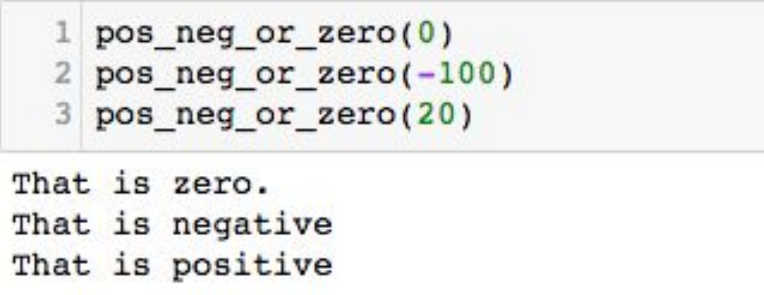
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

This is treated as a single if-statement.

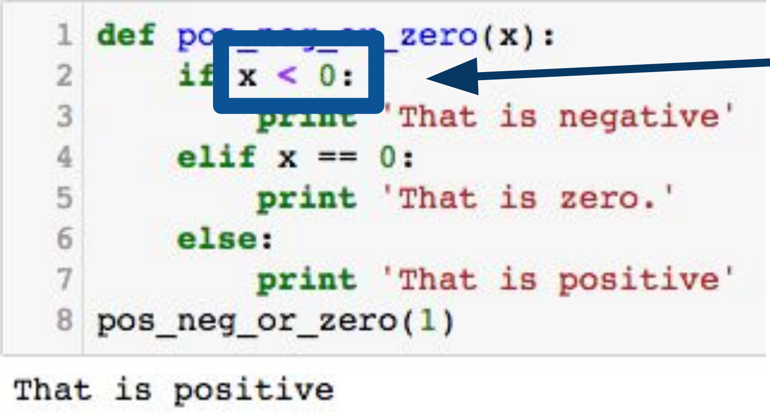


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

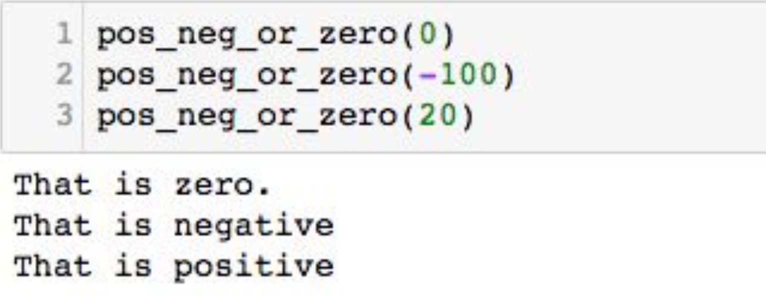
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

If this expression evaluates to `True`...

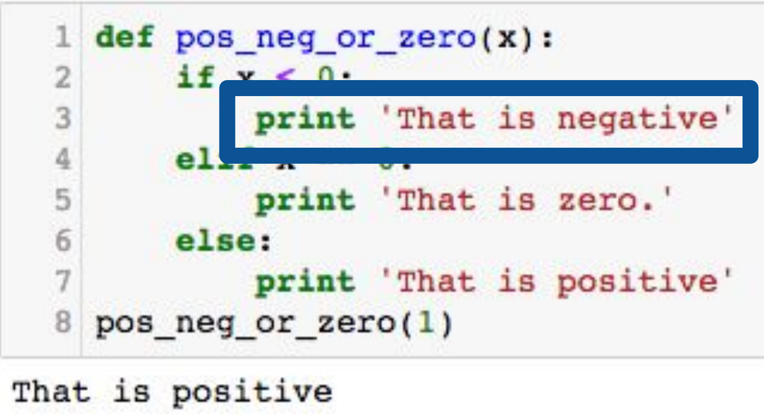


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

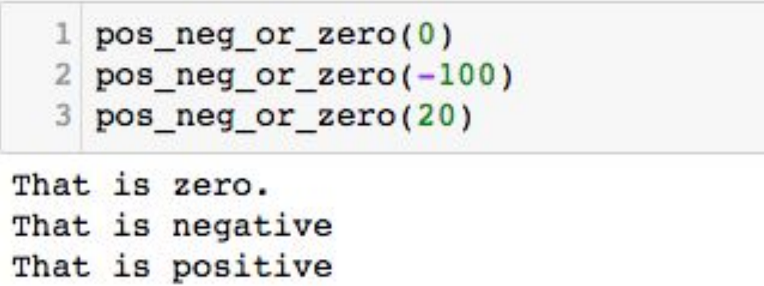
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

...then this block of code is executed...

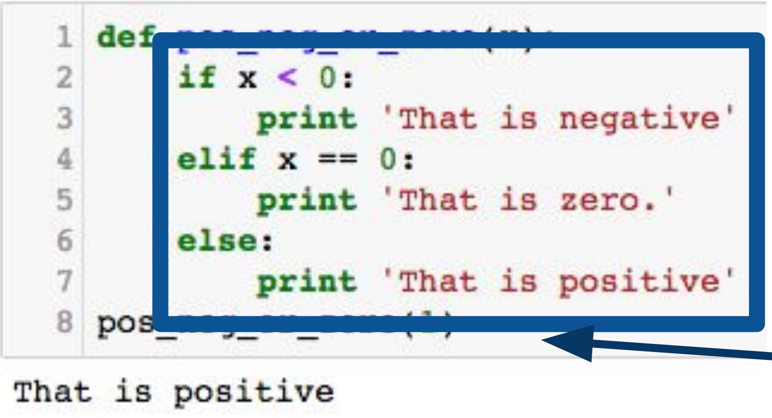


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

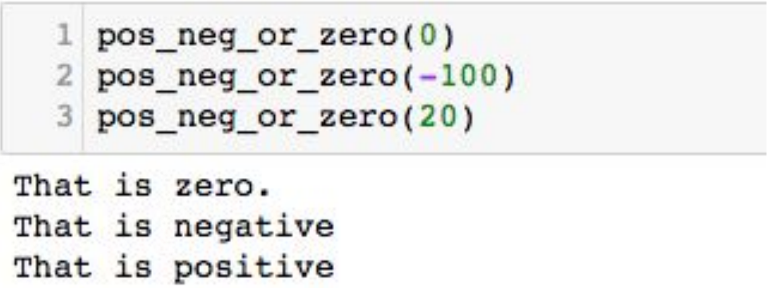
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

...and then Python exits the if-statement

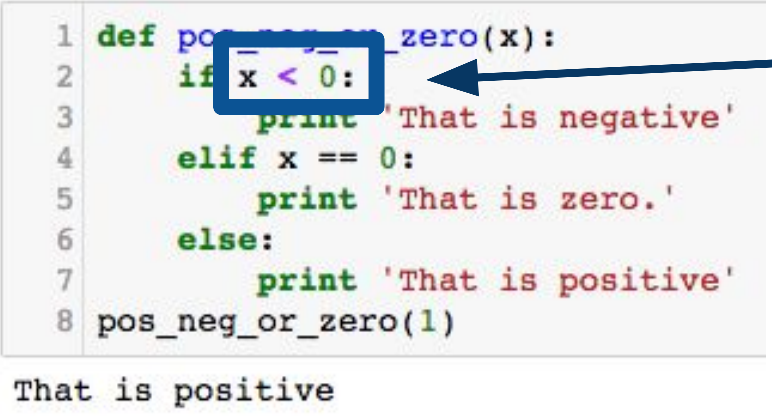


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

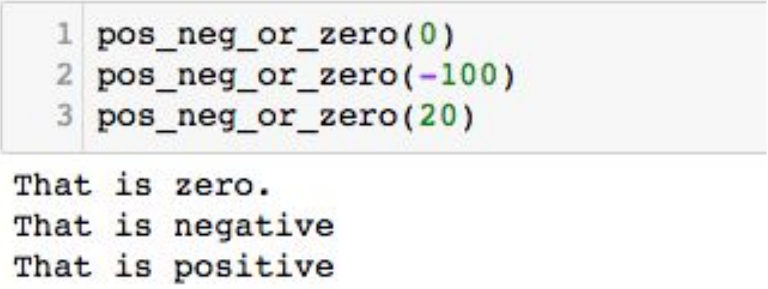
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

If this expression evaluates to `False`...

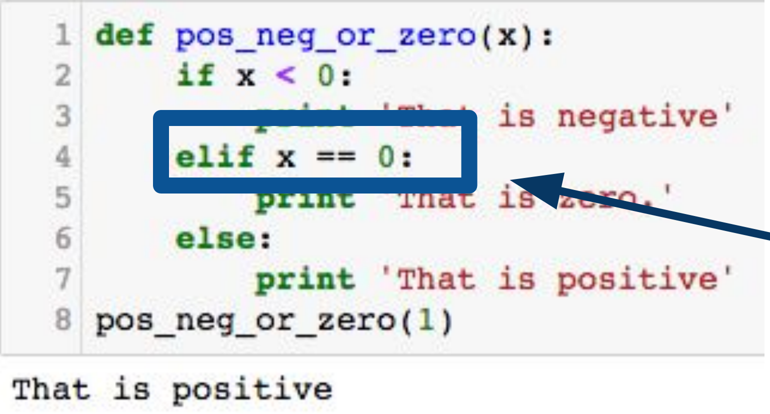


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

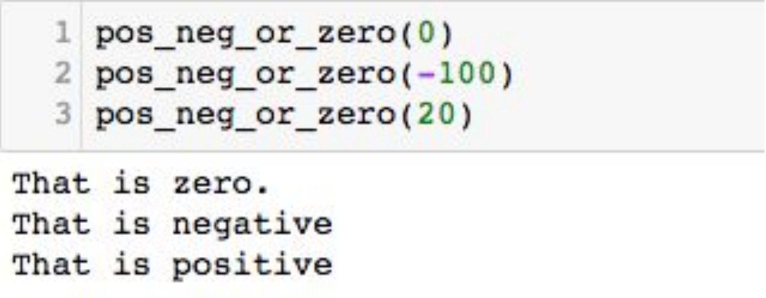
Conditional Expressions

More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```



```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

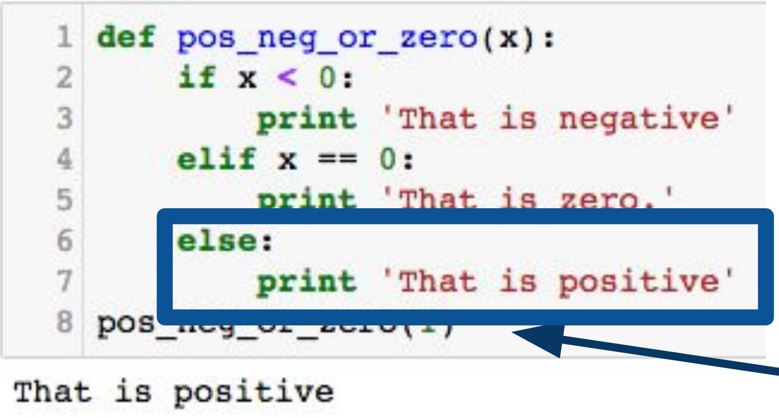
```
That is zero.
That is negative
That is positive
```

Note: `elif` is short for **else if**.

...then we go to the condition. If this condition fails, we go to the next condition, etc.

Conditional Expressions

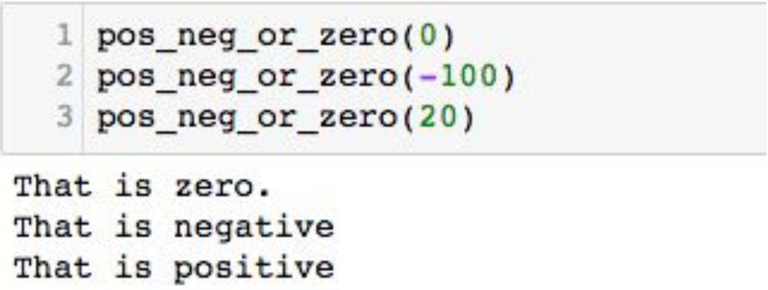
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

If all the other tests fail, we execute the block in the `else` part of the statement.

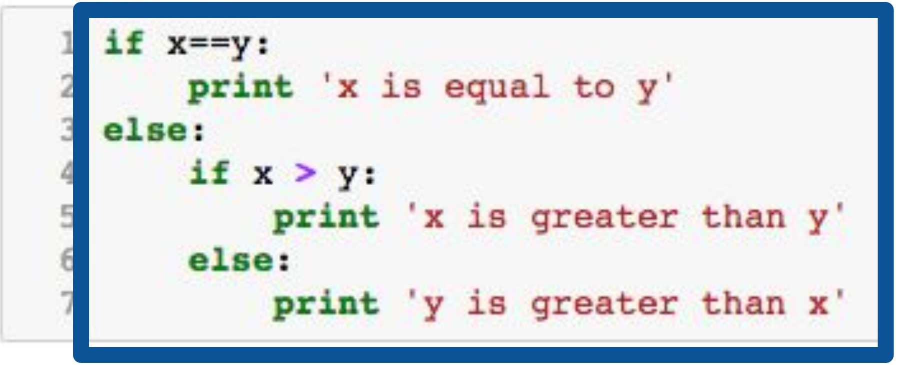


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

Conditionals can also be nested

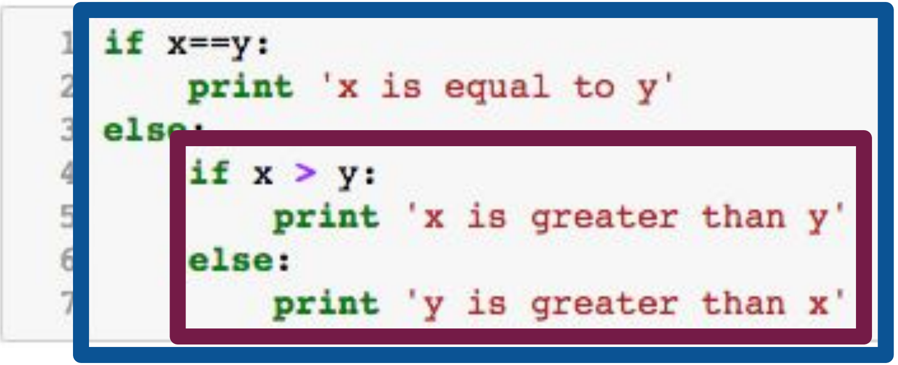


```
if x==y:
    print 'x is equal to y'
else:
    if x > y:
        print 'x is greater than y'
    else:
        print 'y is greater than x'
```

This if-statement...

Conditional Expressions

Conditionals can also be nested



```
if x==y:
    print 'x is equal to y'
else:
    if x > y:
        print 'x is greater than y'
    else:
        print 'y is greater than x'
```

This if-statement...

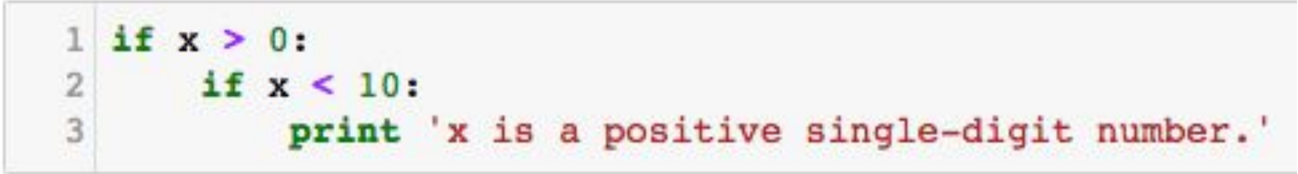
...contains another if-statement.

Conditional Expressions

Often, a nested conditional can be simplified

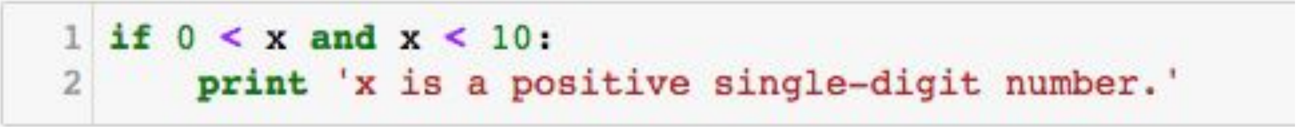
When this is possible, I recommend it for the sake of your sanity, because debugging complicated nested conditionals is tricky!

These two if-statements are equivalent, in that they do the same thing!



```
if x > 0:
    if x < 10:
        print 'x is a positive single-digit number.'
```

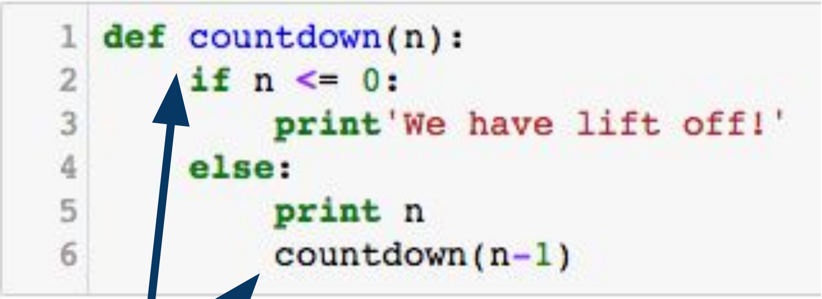
But the second one is (arguably) preferable, as it is simpler to read.



```
if 0 < x and x < 10:
    print 'x is a positive single-digit number.'
```

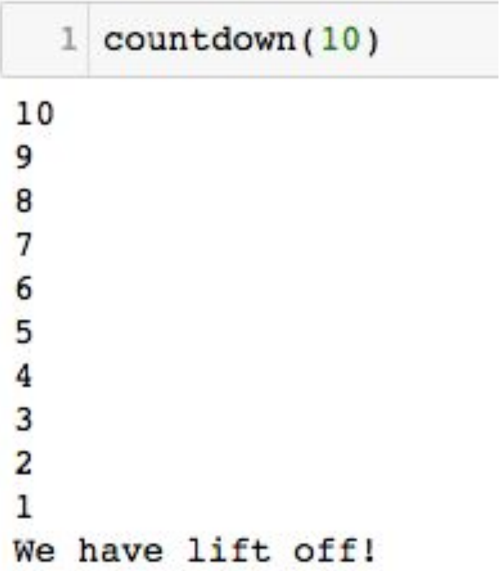
Recursion

A function is a allowed to call itself, in what is termed **recursion**



```
def countdown(n):
    if n <= 0:
        print'We have lift off!'
    else:
        print n
        countdown(n-1)
```

Countdown calls itself!

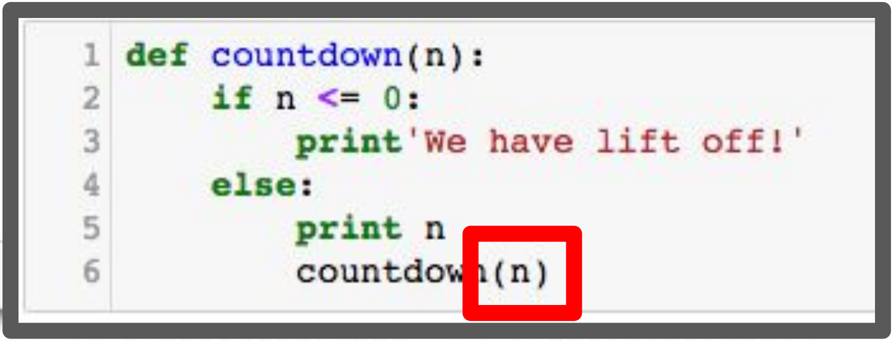


```
countdown(10)
```

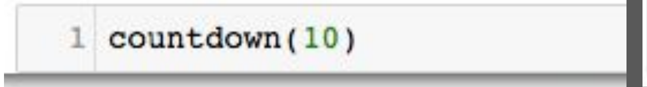
```
10
9
8
7
6
5
4
3
2
1
We have lift off!
```

But the key is that each time it calls itself, it is passing an argument with its value decreased by 1, so eventually, `n <= 0` is true.

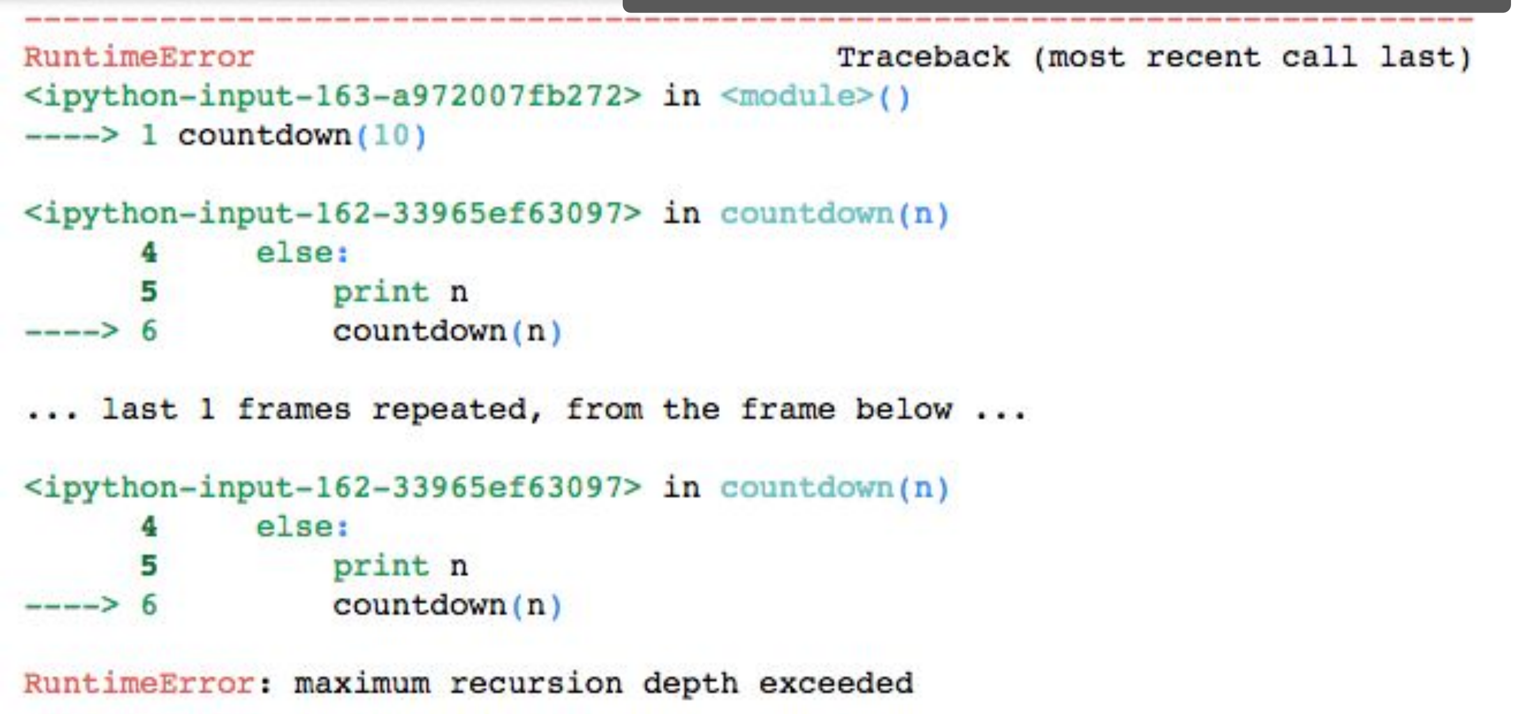
With a small change, we can make it so that `countdown(1)` encounters an **infinite recursion**, in which it repeatedly calls itself.



```
def countdown(n):
    if n <= 0:
        print'We have lift off!'
    else:
        print n
        countdown(n)
```



```
countdown(10)
```

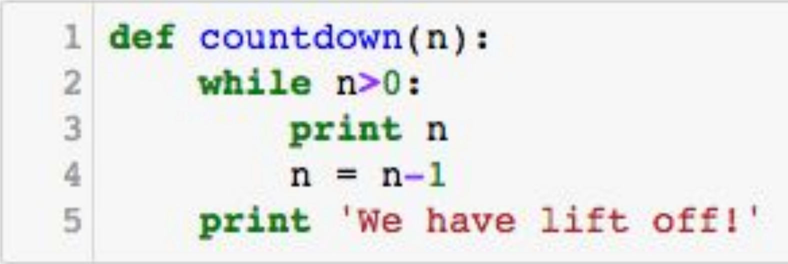


```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-163-a972007fb272> in <module>()
----> 1 countdown(10)

<ipython-input-162-33965ef63097> in countdown(n)
      4     else:
      5         print n
----> 6         countdown(n)

... last 1 frames repeated, from the frame below ...

<ipython-input-162-33965ef63097> in countdown(n)
      4     else:
      5         print n
----> 6         countdown(n)

RuntimeError: maximum recursion depth exceeded
```

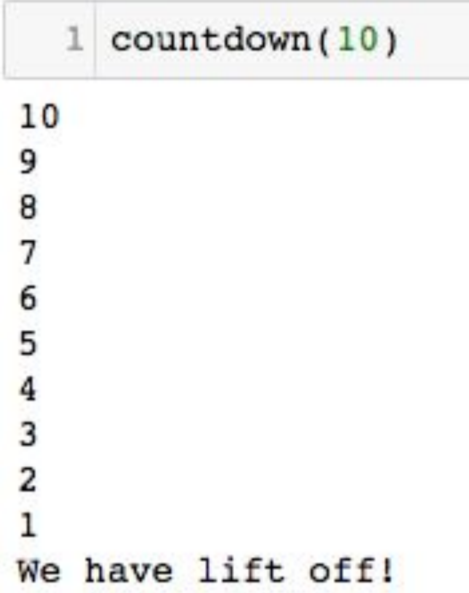
Repeated actions: Iteration

Recursion is the first tool we've seen for performing repeated operations

But there are better tools for the job: `while` and `for` loops.



```
def countdown(n):
    while n>0:
        print n
        n = n-1
    print 'We have lift off!'
```



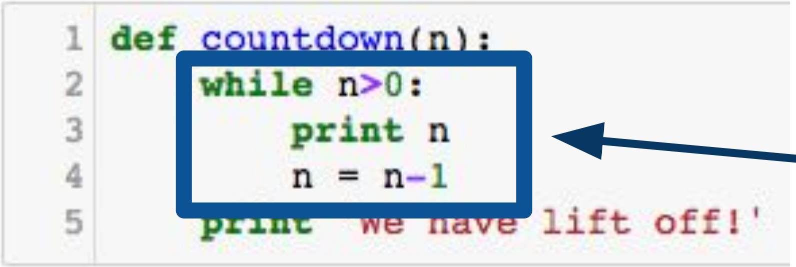
```
countdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
We have lift off!
```

Repeated actions: Iteration

Recursion is the first tool we’ve seen for performing repeated operations

But there are better tools for the job: `while` and `for` loops.



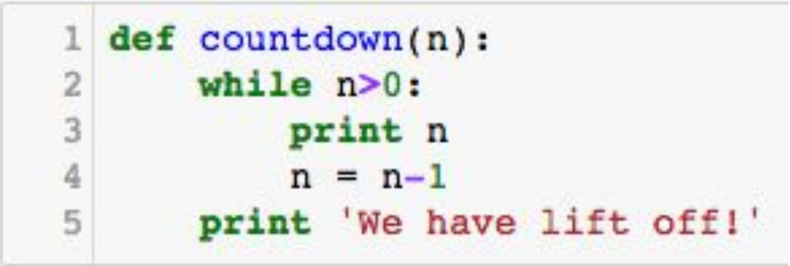
```
def countdown(n):
    while n>0:
        print n
        n = n-1
    print 'We have lift off!'
```

This block specifies a while-loop. So long as the condition is true, Python will run the code in the body of the loop, checking the condition again at the end of each time through.

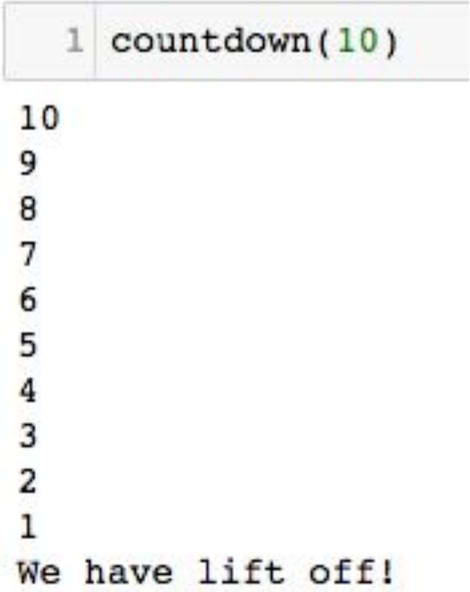
Repeated actions: Iteration

Recursion is the first tool we've seen for performing repeated operations

But there are better tools for the job: `while` and `for` loops.



```python
def countdown(n):
    while n>0:
        print n
        n = n-1
    print 'We have lift off!'
```

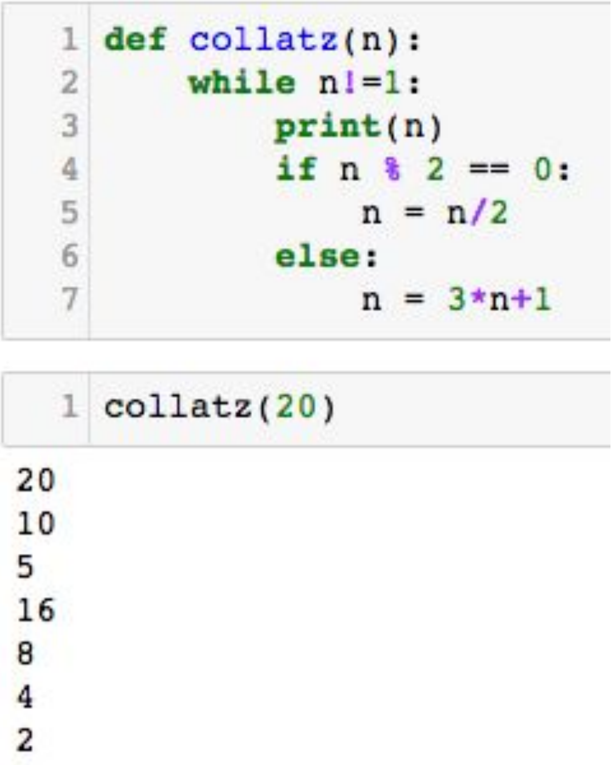


```python
countdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
We have lift off!
```

Warning: Once again, there is a danger of creating an **infinite loop**. If, for example, `n` never gets updated, then when we call `countdown(10)` , the condition `n>0` will always evaluate to `True`, and we will never exit the while-loop.

Repeated actions: Iteration



```
def collatz(n):
    while n!=1:
        print(n)
        if n % 2 == 0:
            n = n/2
        else:
            n = 3*n+1
```

```
collatz(20)
```

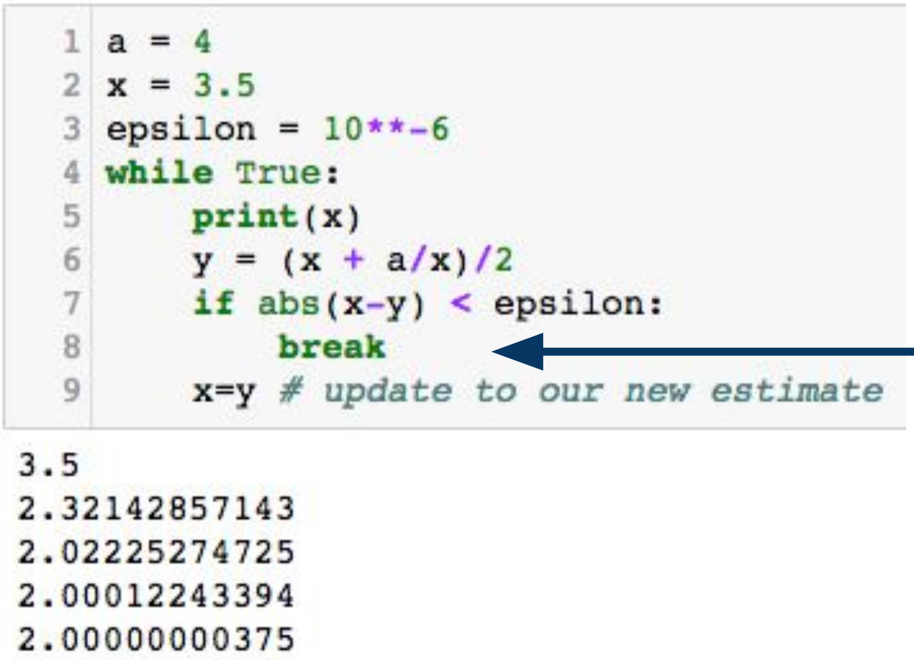
```
20
10
5
16
8
4
2
```

One always wants to try and ensure that a while loop will (eventually) terminate, but it’s not always so easy to know!
https://en.wikipedia.org/wiki/Collatz_conjecture

“Mathematics may not be ready for such problems."
Paul Erdős

Repeated actions: Iteration

We can also terminate a while-loop using the `break` keyword



```python
a = 4
x = 3.5
epsilon = 10**-6
while True:
    print(x)
    y = (x + a/x)/2
    if abs(x-y) < epsilon:
        break
    x=y # update to our new estimate
```

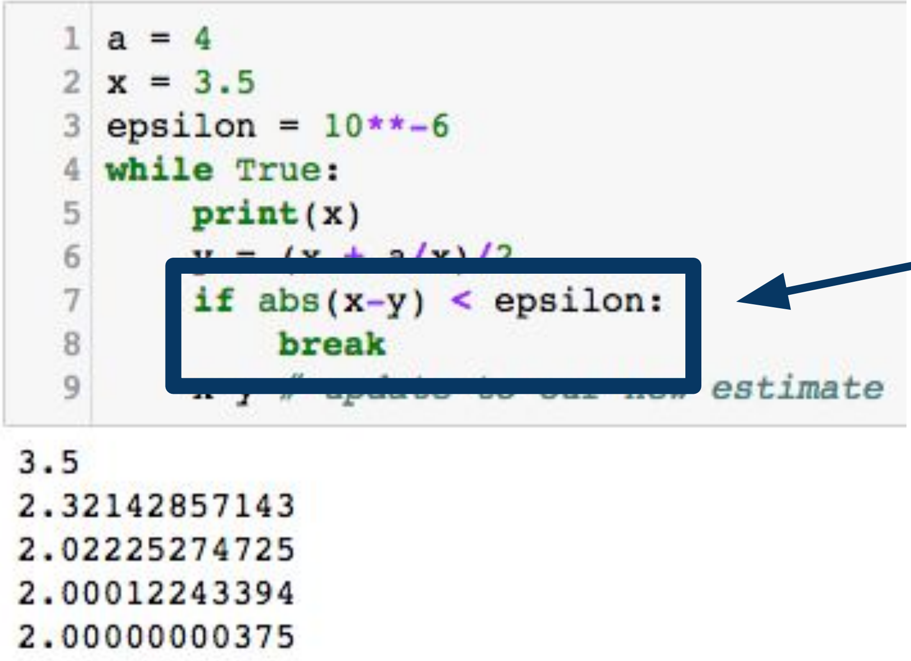
```
3.5
2.32142857143
2.02225274725
2.00012243394
2.00000000375
```

The `break` keyword terminates the current loop when it is called.

Newton-Raphson method: https://en.wikipedia.org/wiki/Newton's_method

Repeated actions: Iteration

We can also terminate a while-loop using the `break` keyword



```
a = 4
x = 3.5
epsilon = 10**-6
while True:
    print(x)
    y = (x + a/x)/2
    if abs(x-y) < epsilon:
        break
    x = y # update to our new estimate
```

```
3.5
2.32142857143
2.02225274725
2.00012243394
2.00000000375
```

Notice that we're not testing for equality here. That's because testing for equality between pairs of floats is dangerous. When I write `x=1/3`, for example, the value of `x` is actually only an approximation to the number 1/3.

Newton-Raphson method: https://en.wikipedia.org/wiki/Newton's_method