

STAT606

Computing for Data Science and Statistics

Lecture 2: Conditionals, Recursion and Iteration

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`

```
1 type(True)
```

```
bool
```

```
1 type(False)
```

```
bool
```

“Boolean” refers to George Boole, an English logician and philosopher.

https://en.wikipedia.org/wiki/George_Boole

Boolean Expressions

Comparison operators available in Python:

```
1 x == y # x is equal to y
2 x != y # x is not equal to y
3 x > y # x is strictly greater than y
4 x < y # x is strictly less than y
5 x >= y # x is greater than or equal to y
6 x <= y # x is less than or equal to y
```

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.

```
1 x = 10
2 y = 20
3 x == y
```

False

```
1 x != y
```

True

```
1 x < x
```

False

```
1 x <= x
```

True

Boolean Expressions

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

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

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

True

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

False

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

True

```
1 not x > 0
```

False

Boolean Expressions

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

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

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

True

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

False

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

True

```
1 not x > 0
```

False

```
1 1 and x > 0
```

True

```
1 0 and x > 0
```

0

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

True

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

''

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

Boolean Expressions: Example

Let's see Boolean expressions in action

```
1 def is_even(n):  
2     # Returns a boolean.  
3     # Returns True if and only if  
4     # n is an even number.  
5     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.

```
1 is_even(0)
```

True

```
1 is_even(1)
```

False

```
1 is_even(8675309)
```

False

```
1 is_even(-3)
```

False

```
1 is_even(12)
```

True

Conditional Expressions

Sometimes we want to do different things depending on certain conditions

```
1 x = 10
2 if x > 0:
3     print 'x is bigger than 0'
4 if x > 1:
5     print 'x is bigger than 1'
6 if x > 100:
7     print 'x is bigger than 100'
8 if x < 100:
9     print 'x is less than 100'
```

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

Note: Oops! I used Python 2 code in this slide! Notice that `print` is being used as a statement, not a function. To run this in Python 3, you'll need to change this code to `read print('x is bigger than 0')`, etc.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions

```
1
2 if x > 0:
3     print 'x is bigger than 0'
4
5 if x > 1:
6     print 'x is bigger than 1'
7
8 if x > 100:
9     print 'x is bigger than 100'
10
11 if x < 100:
12     print 'x is less than 100'
```

This is an **if-statement**.

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

Note: Oops! I used Python 2 code in this slide! Notice that `print` is being used as a statement, not a function. To run this in Python 3, you'll need to change this code to `read print('x is bigger than 0')`, etc.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions

```
1 x = 10
2 if x > 0:
3     print 'x is bigger than 0'
4 if x > 1:
5     print 'x is bigger than 1'
6 if x > 100:
7     print 'x is bigger than 100'
8 if x < 100:
9     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**.

Note: Oops! I used Python 2 code in this slide! Notice that `print` is being used as a statement, not a function. To run this in Python 3, you'll need to change this code to `read print('x is bigger than 0')`, etc.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions

```
1 x = 10
2 if x > 0:
3     print 'x is bigger than 0'
4 if x > 1:
5     print 'x is bigger than 1'
6 if x > 100:
7     print 'x is bigger than 100'
8 if x < 100:
9     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.

Note: Oops! I used Python 2 code in this slide! Notice that `print` is being used as a statement, not a function. To run this in Python 3, you'll need to change this code to `read print('x is bigger than 0')`, etc.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions

```
1 x = 10
2 if x > 0:
3     print 'x is bigger than 0'
4 if x > 1:
5     print 'x is bigger than 1'
6 if x > 100:
7     print 'x is bigger than 100'
8 if x < 100:
9     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.

Note: Oops! I used Python 2 code in this slide! Notice that `print` is being used as a statement, not a function. To run this in Python 3, you'll need to change this code to `read print('x is bigger than 0')`, etc.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions

```
1 x = 10
2 if x > 0:
3     print 'x is bigger than 0'
4 if x > 1:
5     print 'x is bigger than 1'
6 if x > 100:
7     print 'x is bigger than 100'
8 if x < 100:
9     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.

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

```
y is less than 100
```

Conditional Expressions

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

```
1 def pos_neg_or_zero(x):
2     if x < 0:
3         print 'That is negative'
4     elif x == 0:
5         print 'That is zero.'
6     else:
7         print 'That is positive'
8 pos_neg_or_zero(1)
```

That is positive

```
1 pos_neg_or_zero(0)
2 pos_neg_or_zero(-100)
3 pos_neg_or_zero(20)
```

That is zero.
That is negative
That is positive

Note: Oops! Still Python 2! You need to change the print statements to functions to get this to run in Python 3.

Conditional Expressions

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

```
1 def pos_neg_or_zero(x):
2     if x < 0:
3         print 'That is negative'
4     elif x == 0:
5         print 'That is zero.'
6     else:
7         print 'That is positive'
8 pos_neg_or_zero(1)
```

That is positive

```
1 pos_neg_or_zero(0)
2 pos_neg_or_zero(-100)
3 pos_neg_or_zero(20)
```

That is zero.
That is negative
That is positive

← This is treated as a single if-statement.

Note: Oops! Still Python 2! You need to change the print statements to functions to get this to run in Python 3.

Conditional Expressions

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

```
1 def pos_neg_or_zero(x):
2     if x < 0:
3         print 'That is negative'
4     elif x == 0:
5         print 'That is zero.'
6     else:
7         print 'That is positive'
8 pos_neg_or_zero(1)
```

If this expression evaluates to True...

That is positive

```
1 pos_neg_or_zero(0)
2 pos_neg_or_zero(-100)
3 pos_neg_or_zero(20)
```

That is zero.
That is negative
That is positive

Note: Oops! Still Python 2! You need to change the print statements to functions to get this to run in Python 3.

Conditional Expressions

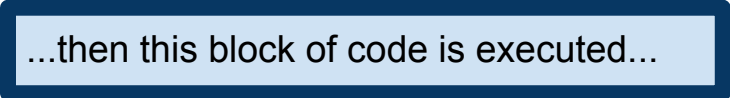
More complicated logic can be handled with **chained conditionals**

```
1 def pos_neg_or_zero(x):
2     if x < 0:
3         print 'That is negative'
4     elif x == 0:
5         print 'That is zero.'
6     else:
7         print 'That is positive'
8 pos_neg_or_zero(1)
```

That is positive

```
1 pos_neg_or_zero(0)
2 pos_neg_or_zero(-100)
3 pos_neg_or_zero(20)
```

That is zero.
That is negative
That is positive



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

Note: Oops! Still Python 2! You need to change the print statements to functions to get this to run in Python 3.

Conditional Expressions

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

```
1 def pos_neg_or_zero(x):  
2     if x < 0:  
3         print 'That is negative'  
4     elif x == 0:  
5         print 'That is zero.'  
6     else:  
7         print 'That is positive'  
8 pos_neg_or_zero(1)
```

That is positive

...and then Python exits the if-statement

```
1 pos_neg_or_zero(0)  
2 pos_neg_or_zero(-100)  
3 pos_neg_or_zero(20)
```

That is zero.
That is negative
That is positive

Note: Oops! Still Python 2! You need to change the print statements to functions to get this to run in Python 3.

Conditional Expressions

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

```
1 def pos_neg_or_zero(x):
2     if x < 0:
3         print 'That is negative'
4     elif x == 0:
5         print 'That is zero.'
6     else:
7         print 'That is positive'
8 pos_neg_or_zero(1)
```

If this expression evaluates to `False`...

That is positive

```
1 pos_neg_or_zero(0)
2 pos_neg_or_zero(-100)
3 pos_neg_or_zero(20)
```

That is zero.
That is negative
That is positive

Note: Oops! Still Python 2! You need to change the print statements to functions to get this to run in Python 3.

Conditional Expressions

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

```
1 def pos_neg_or_zero(x):
2     if x < 0:
3         print 'That is negative'
4     elif x == 0:
5         print 'That is zero.'
6     else:
7         print 'That is positive'
8 pos_neg_or_zero(1)
```

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

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

That is positive

```
1 pos_neg_or_zero(0)
2 pos_neg_or_zero(-100)
3 pos_neg_or_zero(20)
```

That is zero.
That is negative
That is positive

Note: Oops! Still Python 2! You need to change the print statements to functions to get this to run in Python 3.

Conditional Expressions

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

```
1 def pos_neg_or_zero(x):  
2     if x < 0:  
3         print 'That is negative'  
4     elif x == 0:  
5         print 'That is zero.'  
6     else:  
7         print 'That is positive'  
8 pos_neg_or_zero(1)
```

That is positive

```
1 pos_neg_or_zero(0)  
2 pos_neg_or_zero(-100)  
3 pos_neg_or_zero(20)
```

That is zero.
That is negative
That is positive

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

Note: Oops! Still Python 2! You need to change the print statements to functions to get this to run in Python 3.

Conditional Expressions

Conditionals can also be nested

```
1 if x==y:
2     print 'x is equal to y'
3 else:
4     if x > y:
5         print 'x is greater than y'
6     else:
7         print 'y is greater than x'
```

This if-statement...

Note: Oops! Still Python 2! You need to change the print statements to functions to get this to run in Python 3.

Conditional Expressions

Conditionals can also be nested

```
1 if x==y:
2     print 'x is equal to y'
3 else:
4     if x > y:
5         print 'x is greater than y'
6     else:
7         print 'y is greater than x'
```

This if-statement...

...contains another if-statement.

Note: Oops! Still Python 2! You need to change the print statements to functions to get this to run in Python 3.

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!

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

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

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

Recursion

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

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

Countdown calls itself!

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

```
1 countdown(10)
```

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


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

```
1 countdown(10)
```

```
1 def countdown(n):  
2     if n <= 0:  
3         print 'We have lift off!'  
4     else:  
5         print n  
6         countdown(n)
```

```
-----  
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.

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

```
1 countdown(10)
```

10

9

8

7

6

5

4

3

2

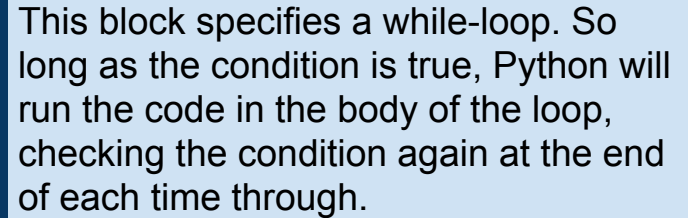
1

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.

```
1 def countdown(n):  
2     while n>0:  
3         print n  
4         n = n-1  
5     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.

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

```
1 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.

```
1 def countdown(n):
2     while n>0:
3         print n
4         n = n-1
5     print '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.

```
1 countdown(10)
```

10

9

8

7

6

5

4

3

2

1

We have lift off!

Repeated actions: Iteration

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

```
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

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

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

```
3.5
2.32142857143
2.02225274725
2.00012243394
2.00000000375
```

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

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

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 $1/3$.

```
3.5
2.32142857143
2.02225274725
2.00012243394
2.00000000375
```

Newton-Raphson method:

https://en.wikipedia.org/wiki/Newton's_method

Aside: floating-point arithmetic

The number $1/3$ has no finite decimal representation, i.e., $1/3 = 0.33333333\dots$
...but our computer uses only finitely many bits to represent a float...

One solution: symbolic computation https://en.wikipedia.org/wiki/Computer_algebra

More practical: **floating-point arithmetic**

https://en.wikipedia.org/wiki/Floating-point_arithmetic

Idea: essentially, pick out a finite set of decimal numbers...

...round variables, solutions, etc to the nearest such number

Drawback: introduces rounding errors e.g., $0.33333333\dots$ becomes 0.333332 .

For more, see Goldberg (1991), *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, **23**(1).