STATS 507
Data Analysis in Python

Lecture 2: Conditionals, Recursion, Iteration and Strings
Boolean Expressions

Boolean expressions evaluate the truth/falsity of a statement.

Python supplies a special Boolean type, `bool`. A variable of type `bool` can be either `True` or `False`.

```
1  type(True)  
    bool

1  type(False) 
    bool
```
Boolean Expressions

Comparison operators available in Python:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>x == y</code> # <code>x</code> is equal to <code>y</code></td>
</tr>
<tr>
<td>2</td>
<td><code>x != y</code> # <code>x</code> is not equal to <code>y</code></td>
</tr>
<tr>
<td>3</td>
<td><code>x &gt; y</code> # <code>x</code> is strictly greater than <code>y</code></td>
</tr>
<tr>
<td>4</td>
<td><code>x &lt; y</code> # <code>x</code> is strictly less than <code>y</code></td>
</tr>
<tr>
<td>5</td>
<td><code>x &gt;= y</code> # <code>x</code> is greater than or equal to <code>y</code></td>
</tr>
<tr>
<td>6</td>
<td><code>x &lt;= y</code> # <code>x</code> is less than or equal to <code>y</code></td>
</tr>
</tbody>
</table>

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`

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>x = 10</code></td>
</tr>
<tr>
<td>2</td>
<td><code>x &lt; 20 and x &gt; 0</code></td>
</tr>
<tr>
<td></td>
<td>True</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>x &gt; 100 and x &gt; 0</code></td>
</tr>
<tr>
<td></td>
<td>False</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>x &gt; 100 or x &gt; 0</code></td>
</tr>
<tr>
<td></td>
<td>True</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>not x &gt; 0</code></td>
</tr>
<tr>
<td></td>
<td>False</td>
</tr>
</tbody>
</table>

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

Sometimes we want to do different things depending on certain conditions

```python
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 bigger than 100
x is less than 100
Conditional Expressions

Sometimes we want to do different things depending on certain conditions

This is an if-statement.
Conditional Expressions

Sometimes we want to do different things depending on certain conditions.

```python
x = ...
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'
```

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.

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

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

```
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.

```python
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'
```

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.

```python
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'
```

\[ x \text{ is bigger than 0} \]
\[ x \text{ is bigger than 1} \]
\[ x \text{ is less than 100} \]

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.

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

\[ y \text{ is less than 100} \]
Conditional Expressions

More complicated logic can be handled with chained conditionals

```python
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

c pos_neg_or_zero(0)
d pos_neg_or_zero(-100)
e 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**

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

These are treated as a single if-statement.
Conditional Expressions

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

```python
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
```

If this expression evaluates to **True**...
Conditional Expressions

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

```python
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
```

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

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

```python
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

```python
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

```python
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)
That is zero.
pos_neg_or_zero(-100)
That is negative
pos_neg_or_zero(20)
That is positive
```

If this expression evaluates to False...
Conditional Expressions

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

```python
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*

```python
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)
That is zero.

pos_neg_or_zero(-100)
That is negative

pos_neg_or_zero(20)
That is positive
```

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

Conditionals can also be nested

```python
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'
```
Conditional Expressions

Conditionals can also be nested

```python
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!

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

A function is 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!

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.

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

```python
def countdown(n):
    while n>0:
        print n
        n = n-1
    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.

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
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.
Repeated actions: Iteration

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](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 
\texttt{break} \hspace{1pt} \texttt{keyword}

\begin{verbatim}
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
\end{verbatim}

The \texttt{break} \hspace{1pt} \texttt{keyword} terminates the current loop when it is called.

Newton-Raphson method:
\url{https://en.wikipedia.org/wiki/Newton's_method}
Repeated actions: Iteration

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

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

Strings are sequences of characters

Python sequences are 0-indexed. The index counts the offset from the beginning of the sequence. So the first letter is the 0-th character of the string.

Note: in some languages, there's a difference between a character and a string of length 1. That is, the character ‘g’ and the string “g” are different data types. In Python, no such difference exists. A character is just a one-character string.
Strings in Python

Strings are sequences of characters.

All Python sequences include a **length** attribute, which is the number of elements in the sequence.

If we try to access an element of the sequence that doesn’t exist, we get an error.

We can also index into a sequence counting from the end.
Strings in Python

We can index into a sequence using an index variable.

...but there's a better way to perform this operation...
Iterations and traversals: for-loops

For-loop provides a more concise way to express the pattern on the right.
Selecting subsequences: slices

A segment of a Python sequence is called a slice.

```python
s = "And now for something completely different"

s[0:7]  # 'And now'
s[12:21]  # 'something'
```

string[m:n] picks out the m-th character to the n-th character, including the m-th character, but not including the n-th character.
Selecting subsequences: slices

A segment of a Python sequence is called a slice.

- `string[:m]` picks out the subsequence starting at 0 through the \((m-1)\)-th character.
- `string[m:]` picks out the subsequence starting at the \(m\)-th character through the end of the sequence.
- Slices also work with negative indexing.
Selecting subsequences: slices

**string[::]** picks out the entire string.

```
1  s = "And now for something completely different"
2  s[:]

'And now for something completely different'
```

**string[x::x]** picks out the x-th through x-th letters, not including the x-th, so this gets the **empty string**.

```
1  s = "And now for something completely different"
2  s[2:2]

''
```
Selecting subsequences: slices

`string[:]` picks out the entire string.

```
1 s = "And now for something completely different"
2 s[:]
'And now for something completely different'
```

`string[x:x]` picks out the x-th through x-th letters, not including the x-th, so this gets the empty string.

```
1 s = "And now for something completely different"
2 s[2:2]
```

The empty string is a string just like any other, but it contains **no letters** and has length 0.
Important concept: immutability

What if I want to change a letter in my string?

```python
mystr = 'goat'
mystr[0] = 'b'
```

Try and assign a different string to a subsequence of a string.

We get an error because strings are immutable. We can't change the value of an existing string.
Important concept: immutability

What if I want to change a letter in my string?

```
1 mystr = 'goat'
2 mystr = 'b'+mystr[1:]
3 mystr

'boat'
```

This avoids the error we saw before because it changes the value of the variable `mystr`, rather than trying to change the contents of a string.
Example: string traversal

```python
def count(word, letter):
    cnt = 0
    for c in word:
        if c == letter:
            cnt = cnt + 1
    return cnt
```

The function `count` makes use of a common pattern, often called a **traversal**. We examine each element of a sequence (i.e., a string), taking some action for each element.

The variable `cnt` keeps a tally of how many times we have seen letter in the string word, so far. We call such a variable a **counter** or an **accumulator**.

```
count('banana', 'a')
3
```

```
count('banana', 'z')
0
```
Python string methods

Python strings provide a number of built-in operations, called **methods**

- `str.upper()` makes all letters in `str` upper case. `str.lower()` is analogous.
- `str.find(sub)` finds the index of the first location of the string `sub` in `str`.
- `str.startswith(sub)` returns `True` if and only if `str` starts with `sub`.
Python string methods

Python strings provide a number of built-in operations, called methods.

```python
1 mystr = 'goat'
2 mystr.upper()

'GOAT'
```

A method is like a function, but it is provided by an object. We’ll learn much more about this later in the semester, but for now, it suffices to know that some data types provide what look like functions (they take arguments and return values), and we call these function-like things methods.

```python
1 'aBcDeFg'.lower()

'abcdefg'
```

```python
1 'banana'.find('na')

2
```

```python
1 'goat'.startswith('go')

True
```

This variable.method() notation is called dot notation, and it is ubiquitous in Python (and many other languages).

Many more Python string methods:
https://docs.python.org/3/library/stdtypes.html#string-methods
Optional arguments: `str.find()`

The `str.find()` method takes **optional arguments**, which specify where in the string to start looking for a match, and the last index to consider for a match.

Find first occurrence of `'na'`, starting from index 3.

Find first occurrence of `'na'`, starting from index 3, and nowhere past 4.

The documentation writes this method as `str.find(sub[, start[, end]])`. Square brackets indicate optional arguments. In this case, brackets also indicate that with two arguments, the second one will be interpreted as the `start` argument. [https://docs.python.org/3/library/stdtypes.html#string-methods](https://docs.python.org/3/library/stdtypes.html#string-methods)
Searching sequences: the `in` keyword

The `in` keyword applies more generally to check whether an object is contained in a sequence. We’ll see more examples of this in the future, but for now, we only need to worry about strings.

- `'a' in 'banana'` returns `True` because `'a'` is a substring of `'banana'`.
- `'z' in 'banana'` returns `False` because `'z'` is not a substring of `'banana'`.
- `'ban' in 'banana'` returns `True` because `'ban'` is a substring of `'banana'`.
- `'anan' in 'banana'` returns `True` because `'anan'` is a substring of `'banana'`.
- `'zoo' in 'banana'` returns `False` because `'zoo'` is not a substring of `'banana'`.

Importantly, we can check for a whole substring, making this very similar to `str.find()`.

The `in` keyword returns `True` if `x` occurs as a substring of `y`, and `False` otherwise.
String Comparison

Sometimes we want to check if two strings are equal

Use the equality operator (==), just like for comparing numbers.

Strings have to match exactly. Substring is not enough!
String Comparison

Sometimes we want to check if two strings are equal.

Use the equality operator (==), just like for comparing numbers.

Strings have to match exactly. Substring is not enough!

If we can compare strings with equality, we should be able to compare them with inequalities, too...
String Comparison

We can also compare words under alphabetical ordering.

```
1 'cat' < 'dog'
True

1 'cat' >= 'dog'
False

1 'dog' < 'doge'
True

1 '' < 'goat'
True

1 'l' < 'a'
True
```

- Words earlier in the dictionary are “smaller” than words later in the dictionary.
- The empty string '' comes first in the ordering.
- Strings including numbers, symbols, etc. are also ordered.
String Comparison

**Important:** upper case and lower case letters ordered differently!

<table>
<thead>
<tr>
<th></th>
<th>'Cat' == 'cat'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>False</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>'cat' &gt; 'Cat'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>True</td>
</tr>
</tbody>
</table>

Upper case letters are ordered before lower case letters.

For more information: [https://docs.python.org/3/library/stdtypes.html#comparisons](https://docs.python.org/3/library/stdtypes.html#comparisons)

For much more information: [https://docs.python.org/3/library/operator.html?highlight=equallity](https://docs.python.org/3/library/operator.html?highlight=equallity)
Python Lists

Strings in Python are “sequences of characters”

But what if I want a sequence of something else?
   A vector would be naturally represented as a sequence of numbers
   A class roster might be represented as a sequence of strings

Python lists are sequences whose values can be of any data type
   We call these list entries the elements of the list