INTL/QMBU 472 / CSSM 502: Advanced Data Analysis in Python

Syntax

David Carlson

September 30, 2021

Syntax

- Object types
  - String
  - Int
  - Float
  - List
  - Tuple
  - Dictionary
- Conditionals
- Loop
- Functions

Strings

- Any group of characters recognized as text.

Strings

- Any group of characters recognized as text.
- Written between single quotes, double quotes or triple quotes.

Strings

- Any group of characters recognized as text.
- Written between single quotes, double quotes or triple quotes.

```
>>> name='David'
>>> age='34'
>>> intro="Hi my name is "+name+".\nI'm "+age+" years old.
>>> intro
>>> print(intro)
>>> new_intro = """Hello!
... I'm David.
... What's up?"""
>>> new_intro
>>> print(new_intro)
```

String

- You can call any character in the string.

String

- You can call any character in the string.

```
>>> intro[0]
>>> intro[1]
>>> intro[3]
```

String

- You can call any character in the string.

```
>>> intro[0]
>>> intro[1]
>>> intro[3]
```

- Strings are immutable.

String

- You can call any character in the string.

```
>>> intro[0]
>>> intro[1]
>>> intro[3]
```

- Strings are immutable.
- But you can split a string into words.

String

- You can call any character in the string.

```
>>> intro[0]
>>> intro[1]
>>> intro[3]
```

- Strings are immutable.
- But you can split a string into words.

```
>>> intro.split()
```

String

- You can call any character in the string.

```
>>> intro[0]
>>> intro[1]
>>> intro[3]
```

- Strings are immutable.
- But you can split a string into words.

```
>>> intro.split()
```

- Or into any other chunks using a character.

String

- You can call any character in the string.

```
>>> intro[0]
>>> intro[1]
>>> intro[3]
```

- Strings are immutable.
- But you can split a string into words.

```
>>> intro.split()
```

- Or into any other chunks using a character.

```
>>> new_intro.split('\n')
```

String

- You can call any character in the string.

```
>>> intro[0]
>>> intro[1]
>>> intro[3]
```

- Strings are immutable.
- But you can split a string into words.

```
>>> intro.split()
```

- Or into any other chunks using a character.

```
>>> new_intro.split('\n')
```

String

- Run this code. What is happening?

String

- Run this code. What is happening?

```
>>> intro[2:]
>>> intro[-2:]
>>> intro[:2]
>>> intro[:-2]
>>> intro[::2]
>>> intro[::-2]
>>> intro[::3]
```

String

- It requires a little more work to split a string into letters.

String

- It requires a little more work to split a string into letters.

```
>>> [letter for letter in name]
>>> [letter for letter in intro]
```

String

- It requires a little more work to split a string into letters.

```
>>> [letter for letter in name]
>>> [letter for letter in intro]
```

- Let's combine them again.

String

- It requires a little more work to split a string into letters.

```
>>> [letter for letter in name]
>>> [letter for letter in intro]
```

- Let's combine them again.

```
>>> myletters=[letter for letter in intro]
>>> ''.join(myletters)
>>> '\n'.join(myletters)
```

Int

- Integers.

Int

- Integers.
- You can do mathematical operations using these.
  - Usual suspects: + - * /
  - Exponentiate: **
  - Remainder: %
  - Whole division: //

Int

- Integers.
- You can do mathematical operations using these.
  - Usual suspects: + - * /
  - Exponentiate: **
  - Remainder: %
  - Whole division: //

```
>>> whole=5//3
>>> remainder=5%3
>>> """Five divided by three is %d
... and %d fifths""" % (whole, remainder)
```

Int

- Integers.
- You can do mathematical operations using these.
  - Usual suspects: + - * /
  - Exponentiate: **
  - Remainder: %
  - Whole division: //

```
>>> whole=5//3
>>> remainder=5%3
>>> """Five divided by three is %d
... and %d fifths""" % (whole, remainder)
```

- You can assign numbers using different operators.

Int

- Integers.
- You can do mathematical operations using these.
  - Usual suspects: + - * /
  - Exponentiate: **
  - Remainder: %
  - Whole division: //

```
>>> whole=5//3
>>> remainder=5%3
>>> """Five divided by three is %d
... and %d fifths""" % (whole, remainder)
```

- You can assign numbers using different operators.

```
>>> five=5
>>> five+=1
>>> five
>>> five/=3
>>> five
>>> five-=2
>>> five
```

Float

- Real numbers.

Float

- Real numbers.
- Written by adding the decimal to an integer.

Float

- Real numbers.
- Written by adding the decimal to an integer.

```
>>> 12.0/5
>>> float(7)
>>> type(2.*8)
```

List

- Collection of any type objects – even lists

List

- Collection of any type objects – even lists

```
>>> myletters
>>> type(myletters)
```

List

- Collection of any type objects – even lists

```
>>> myletters
>>> type(myletters)
```

- Lists can be changed, and include multiple object types

List

- Collection of any type objects – even lists

```
>>> myletters
>>> type(myletters)
```

- Lists can be changed, and include multiple object types

```
>>> myletters.append(5)
>>> myletters[-1]
>>> type(myletters[-1])
>>> myletters[0]='Orange'
```

List

- Collection of any type objects – even lists

```
>>> myletters
>>> type(myletters)
```

- Lists can be changed, and include multiple object types

```
>>> myletters.append(5)
>>> myletters[-1]
>>> type(myletters[-1])
>>> myletters[0]='Orange'
```

- Indexing starts at 0!

List

- Collection of any type objects – even lists

```
>>> myletters
>>> type(myletters)
```

- Lists can be changed, and include multiple object types

```
>>> myletters.append(5)
>>> myletters[-1]
>>> type(myletters[-1])
>>> myletters[0]='Orange'
```

- Indexing starts at 0!

```
>>> myletters[len(myletters)]
```

List

- Collection of any type objects – even lists

```
>>> myletters
>>> type(myletters)
```

- Lists can be changed, and include multiple object types

```
>>> myletters.append(5)
>>> myletters[-1]
>>> type(myletters[-1])
>>> myletters[0]='Orange'
```

- Indexing starts at 0!

```
>>> myletters[len(myletters)]
```

- You can insert into any position

```
>>> myletters.insert(2, '!')
```

List

- Collection of any type objects – even lists

```
>>> myletters
>>> type(myletters)
```

- Lists can be changed, and include multiple object types

```
>>> myletters.append(5)
>>> myletters[-1]
>>> type(myletters[-1])
>>> myletters[0]='Orange'
```

- Indexing starts at 0!

```
>>> myletters[len(myletters)]
```

- You can insert into any position

```
>>> myletters.insert(2, '!')
```

- And remove from any position

List

- Collection of any type objects – even lists

```
>>> myletters
>>> type(myletters)
```

- Lists can be changed, and include multiple object types

```
>>> myletters.append(5)
>>> myletters[-1]
>>> type(myletters[-1])
>>> myletters[0]='Orange'
```

- Indexing starts at 0!

```
>>> myletters[len(myletters)]
```

- You can insert into any position

```
>>> myletters.insert(2, '!')
```

- And remove from any position

```
>>> myletters.pop(1)
```

Tuples

- Tuples are like lists – combination of any objects

Tuples

- Tuples are like lists – combination of any objects
- But are immutable

Tuples

- Tuples are like lists – combination of any objects
- But are immutable
- Not very common, but very useful sometimes

Tuples

- Tuples are like lists – combination of any objects
- But are immutable
- Not very common, but very useful sometimes

```
>>> tup=(1,6,5,'Apple')
>>> tup[1]
>>> tup[1]=9
>>> tup.append(9)
```

Dictionary

- It is what it sounds like.

Dictionary

- It is what it sounds like.
- Here is how you create one.

Dictionary

- It is what it sounds like.
- Here is how you create one.

```
>>> myDict={'name':'David',
... 'last_name':'Carlson', 'age':34}
```

- Unlike lists, there is no order to elements.

Dictionary

- It is what it sounds like.
- Here is how you create one.

```
>>> myDict={'name':'David',
... 'last_name':'Carlson', 'age':34}
```

- Unlike lists, there is no order to elements.
- You call elements using keys.

Dictionary

- It is what it sounds like.
- Here is how you create one.

```
>>> myDict={'name':'David',
... 'last_name':'Carlson', 'age':34}
```

- Unlike lists, there is no order to elements.
- You call elements using keys.

```
>>> myDict
>>> myDict.keys()
>>> myDict.values()
>>> myDict['last_name']
>>> myDict['middle_name']='George'
```

Dictionary

- It is what it sounds like.
- Here is how you create one.

```
>>> myDict={'name':'David',
... 'last_name':'Carlson', 'age':34}
```

- Unlike lists, there is no order to elements.
- You call elements using keys.

```
>>> myDict
>>> myDict.keys()
>>> myDict.values()
>>> myDict['last_name']
>>> myDict['middle_name']='George'
```

- These are particularly useful when we start defining classes (next class)

Conditionals

- Perform an operation (or several) if condition is met (or not)

Conditionals

- Perform an operation (or several) if condition is met (or not)

```
>>> x=2
>>> if x==1:
...     print('x is one')
... elif x==2:
...     print('x is two')
... else:
...     print('x is neither one nor two')
```

Conditionals

- Perform an operation (or several) if condition is met (or not)

```
>>> x=2
>>> if x==1:
...     print('x is one')
... elif x==2:
...     print('x is two')
... else:
...     print('x is neither one nor two')
```

- Can be conditions ($<$, $>$, $<=$, $>=$, $==$, $!=$) or boolean (True or False)

Conditionals

- Perform an operation (or several) if condition is met (or not)

```
>>> x=2
>>> if x==1:
...     print('x is one')
... elif x==2:
...     print('x is two')
... else:
...     print('x is neither one nor two')
```

- Can be conditions ($<$, $>$, $<=$, $>=$, $==$, $!=$) or boolean (True or False)
- Multiple lines of code:

Conditionals

- Perform an operation (or several) if condition is met (or not)

```
>>> x=2
>>> if x==1:
...     print('x is one')
... elif x==2:
...     print('x is two')
... else:
...     print('x is neither one nor two')
```

- Can be conditions ($<$, $>$, $<=$, $>=$, $==$, $!=$) or boolean (True or False)
- Multiple lines of code:
  - Indentation matters!

Conditionals

- Perform an operation (or several) if condition is met (or not)

```
>>> x=2
>>> if x==1:
...     print('x is one')
... elif x==2:
...     print('x is two')
... else:
...     print('x is neither one nor two')
```

- Can be conditions ($<$, $>$, $<=$, $>=$, $==$, $!=$) or boolean (True or False)
- Multiple lines of code:
  - Indentation matters!
  - Consistency is important, but exactly 4 spaces is 'Pythonic'

Conditionals

- Perform an operation (or several) if condition is met (or not)

```
>>> x=2
>>> if x==1:
...     print('x is one')
... elif x==2:
...     print('x is two')
... else:
...     print('x is neither one nor two')
```

- Can be conditions ($<$, $>$, $<=$, $>=$, $==$, $!=$) or boolean (True or False)
- Multiple lines of code:
  - Indentation matters!
  - Consistency is important, but exactly 4 spaces is 'Pythonic'
  - Will cause errors

Conditionals

- Perform an operation (or several) if condition is met (or not)

```
>>> x=2
>>> if x==1:
...     print('x is one')
... elif x==2:
...     print('x is two')
... else:
...     print('x is neither one nor two')
```

- Can be conditions ($<$, $>$, $<=$, $>=$, $==$, $!=$) or boolean (True or False)
- Multiple lines of code:
  - Indentation matters!
  - Consistency is important, but exactly 4 spaces is 'Pythonic'
  - Will cause errors
  - Even an empty line with spaces can cause errors

Conditionals

- Perform an operation (or several) if condition is met (or not)

```
>>> x=2
>>> if x==1:
...     print('x is one')
... elif x==2:
...     print('x is two')
... else:
...     print('x is neither one nor two')
```

- Can be conditions ($<$, $>$, $<=$, $>=$, $==$, $!=$) or boolean (True or False)
- Multiple lines of code:
  - Indentation matters!
  - Consistency is important, but exactly 4 spaces is 'Pythonic'
  - Will cause errors
  - Even an empty line with spaces can cause errors

Loops

- Two types of loops: for and while

Loops

- Two types of loops: for and while
- for loop: loops over some list

Loops

- Two types of loops: for and while
- for loop: loops over some list
- while loop: loops while condition is true

Loops

- Two types of loops: for and while
- for loop: loops over some list
- while loop: loops while condition is true
- Can nest loops (and conditionals, etc.)

Loops

- Two types of loops: for and while
- for loop: loops over some list
- while loop: loops while condition is true
- Can nest loops (and conditionals, etc.)

```
>>> even_numbers=[]
>>> for i in range(1,10):
...     if i%2==0:
...         even_numbers.append(i)
...
>>> for letter in 'word': print(letter)
...
>>> sum([.05**i for i in range(1,10)])
>>> while len(myletters)>1:
...     myletters.pop()
...
```

Quick exercise

- Write code that saves the first ten numbers of the Fibonacci sequence to a list:

Quick exercise

- Write code that saves the first ten numbers of the Fibonacci sequence to a list:
  - With a for loop

Quick exercise

- Write code that saves the first ten numbers of the Fibonacci sequence to a list:
  - With a for loop
  - With a while loop

Quick exercise

- Write code that saves the first ten numbers of the Fibonacci sequence to a list:
  - With a for loop
  - With a while loop
- A while loop can always do what a for loop does, but syntax is simpler

Functions

- They help write cleaner code.

Functions

- They help write cleaner code.
- Keep them simple.

Functions

- They help write cleaner code.
- Keep them simple.
- You can return any type of object.

Functions

- They help write cleaner code.
- Keep them simple.
- You can return any type of object.
- Don't forget to add `return` for output.

Functions

- They help write cleaner code.
- Keep them simple.
- You can return any type of object.
- Don't forget to add `return` for output.

```
>>> def addSquares(x,y):
...     return x**2+y**2
...
>>> addSquares(3,4)
```

Functions

- They help write cleaner code.
- Keep them simple.
- You can return any type of object.
- Don't forget to add `return` for output.

```
>>> def addSquares(x,y):
...     return x**2+y**2
...
>>> addSquares(3,4)
```

- Change the Fibonacci code to find first n numbers of sequence