CSCI 303

Introduction to Data Science

3 - Python Sequence Types

In [1]:

```python
x=range(10)
x
```

Out[1]:

```
range(0, 10)
```

Preview

In [2]:

```python
x = range(10)                    # range object
y = [(n, n * n) for n in x]      # list comprehension
for a, asq in y:                 # for loop (w/variable unpacking)
    print(a, 'squared is', asq)
```

```
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
```

Sequence Types

- strings:
  - `'single quotes or'`
  - `"double quotes allowed"`
- lists: `[1, 1.0, 'one']`
- tuples: `(3.1415, True, "hello")`

Lists

Like an array in many languages

- Indexed sequence of values
- Zero-based indexing

However, can contain mixed types.

Basic operations via square brackets, similar to C++:

In [3]:

```
arr = ['a', 'b', 'c']
print(arr[0])
print("Hi")
```

```
a
Hi
```

You can also replace the value of an indicy:

In [4]:

```
arr[1] = 'x'
print(arr)
```

```
['a', 'x', 'c']
```

Indices can also be negative, in which case they start from the right:

In [5]:

```
print(arr[-1])
```

```
c
```

List Slices

Slicing is a mechanism to obtain a sub-sequence from a sequence:

`arr[n:m]` means "give me the sub-sequence of arr which starts at index n and ends at index m - 1"

Try it:

In [6]:

```
arr = [0,1,2,3,4,5,6,7,8,9,10]
# note we don't need to always use print();
# Jupyter will always print the last value produced.
# Also, # starts a comment
arr[1:3]
```

Out[6]:

```
[1, 2]
```

More Slicing

You can also slice with negative indices:

In [7]:

```
arr = [0,1,2,3,4,5,6,7,8,9,10]

# will output the value at the fourth index (inclusive) through the value at the
arr[4:-2]
```

Out[7]:

```
[4, 5, 6, 7, 8]
```

You can also omit either or both of the indices; the first index defaults to zero, the second to the length of the sequence:

In [8]:

```
# the first five values in the list
arr[:5]
```

Out[8]:

```
[0, 1, 2, 3, 4]
```

In [9]:

```
# the sixth value until the end of the list
arr[5:]
```

Out[9]:

```
[5, 6, 7, 8, 9, 10]
```

You can optionally slice using an increment, to skip over values in a list:

In [10]:

```
# every third value from the start of the list until the end of it
arr[0:10:3]  # or just arr[::3]
```

Out[10]:

```
[0, 3, 6, 9]
```

Other Sequences

Indexing and slicing also work on strings and tuples:

In [11]:

```
s = 'Data Science'
s[5:]
```

Out[11]:

```
'Science'
```

In [12]:

```
t = ('a', 'b', 'c')
t[1]
```

Out[12]:

```
'b'
```

However, there are some differences. In particular, strings and tuples are *immutable* types, so you cannot change a string or tuple value once created (although you can create new strings and tuples using slices and concatenation).

Lists are Mutable

Unlike strings and tuples, you can modify list objects in various ways:

In [13]:

```
arr = [0,1,2,3,4,5,6,7,8,9,10]
arr[0] = 17
arr
```

Out[13]:

```
[17, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [14]:

```
arr.append(11)
arr
```

Out[14]:

```
[17, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Using slicing, you can modify lists in some very flexible ways, including inserting and deleting subsequences:

In [15]:

```
# inserts 0 into indicies 4,5 and 6.
arr[4:7] = [0,0,0]
arr
```

Out[15]:

```
[17, 1, 2, 3, 0, 0, 0, 7, 8, 9, 10, 11]
```

In [16]:

```
# deletes indicies 4,5, and 6.
arr[4:7] = []
arr
```

Out[16]:

```
[17, 1, 2, 3, 7, 8, 9, 10, 11]
```

In [17]:

```
arr[3:3] = ['a','b','c','d']
arr
```

Out[17]:

```
[17, 1, 2, 'a', 'b', 'c', 'd', 3, 7, 8, 9, 10, 11]
```

del

The operator `del` can also be used to remove elements by index or slice from a list:

In [18]:

```
arr = [0,1,2,3,4,5,6,7,8,9,10]
del arr[4]
arr
```

Out[18]:

```
[0, 1, 2, 3, 5, 6, 7, 8, 9, 10]
```

In [19]:

```
arr = [0,1,2,3,4,5,6,7,8,9,10]
del arr[::2]
arr
```

Out[19]:

```
[1, 3, 5, 7, 9]
```

Slicing: A Final Note

When in an expression (i.e., **not** on the LHS of an assignment), slices of basic Python types are always *copies*. E.g.,

In [20]:

```
arr = [0,1,2,3,4,5]
sl = arr[1:3]
sl[0] = 17
print(arr, sl)
```

```
[0, 1, 2, 3, 4, 5] [17, 2]
```

As we'll see, NumPy arrays have a different behavior.

List Methods

Lists have a number of additional methods that you may find useful, some of which are listed below. For the examples, assume `a = [1,7,4]`:

method	example	result
append	a.append(3)	a = [1,7,4,3]
extend	a.extend([4,5,6])	a = [1,7,4,4,5,6]
sort	a.sort()	a = [1,4,7]
reverse	a.reverse()	a = [4,7,1]

Do `help(list)` for full documentation.

Miscellaneous Sequence Operations

The built-in function `len` gives you the size of a sequence:

In [21]:

```
len("Hello, World!")
```

Out[21]:

```
13
```

Also try `max` and `min` :

In [22]:

```
max([8,4,17,3])
```

Out[22]:

```
17
```

Concatenation via `+` works on sequences:

In [23]:

```
('a','b','c') + ('d', 'e', 'f')
```

Out[23]:

```
('a', 'b', 'c', 'd', 'e', 'f')
```

The `*` operator concatenates repetitions of a sequence:

In [24]:

```
print("abc" * 3)
print([1,2,3] * 2)
```

```
abcabcabc
[1, 2, 3, 1, 2, 3]
```

Containment is tested using `in` and `not in` as binary operators:

In [25]:

```
x = 42
a = [1,2,3,4,5]
x in a
```

Out[25]:

```
False
```

In [26]:

```
x not in a
```

Out[26]:

```
True
```

Variable Unpacking

Given an expression resulting in a list, tuple, or similar object, you can break the object into its parts by assigning to a comma-separated list of variables:

In [27]:

```python
record = [1234, 'apple', 0.45]
sku, description, price = record
print(sku, description, price)
```

```
1234 apple 0.45
```

For Loop

`for` loops in Python always iterate over an object representing (or representable as) a sequence: objects that are determined to be *iterable*.

Some types of iterable objects:

- lists, strings, tuples
- files
- *range* objects
- database query results

For Loop Syntax

Syntax:

```
for <var> in <iterable object>:
   <statements>
```

Note again, indentation is used to determine the statement block.

For Example

In [28]:

```python
import math
x = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
for n in x:
    root = math.sqrt(n)
    fracpart, intpart = math.modf(root)
    if fracpart == 0.0:
        print(n, 'is a perfect square')
```

```
0 is a perfect square
1 is a perfect square
4 is a perfect square
9 is a perfect square
16 is a perfect square
```

Wondering what is going on above? Remember you can use ? or help() to get more info!

In [29]:

```
math.modf?
```

For Example with Unpacking

Try this:

In [30]:

```
pairs = [(1,2), (3,4), (5,6)]
for x, y in pairs:
    print(x * y)
```

```
2
12
30
```

Range

A range is an object representing an evenly spaced sequence of integers.

A range object doesn't store its values, it produces them on demand.

Example:

In [31]:

```
range(10)
```

Out[31]:

```
range(0, 10)
```

In [32]:

```
for x in range(10):
    print(x, end = ' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

The `range` constructor can take in an optional start value (default is zero), a mandatory end value, and an optional increment (default is 1), in that order. If two values are provided they are interpreted as start and end values.

Examples:

In [33]:

```
for x in range(3,7):
    print(x, end = ' ')
```

```
3 4 5 6
```

In [34]:

```
for x in range(0,10,2):
    print(x, end = ' ')
```

```
0 2 4 6 8
```

In [35]:

```
for x in range(10,0,-1):
    print(x, end = ' ')
```

```
10 9 8 7 6 5 4 3 2 1
```

For, Range, and Python Style

Note that this is considered very "un-pythonic":

In [36]:

```python
arr = ["one", "two", "three"]
for i in range(len(arr)):
    print(arr[i])
```

```
one
two
three
```

It is strongly preferred to simply loop on the list:

In [37]:

```python
for s in arr:
    print(s)
```

```
one
two
three
```

List Comprehensions

Compare the following:

In [38]:

```python
squares = []
for x in range(5):
    squares.append(x * x)
squares
```

Out[38]:

```
[0, 1, 4, 9, 16]
```

In [39]:

```python
squares = [x * x for x in range(5)]
squares
```

Out[39]:

```
[0, 1, 4, 9, 16]
```

The basic syntax is

```
[<expr> for <var> in <obj>]
```

which results in a new list built of each evaluation of `<expr>` .

The expression can be anything (and doesn't have to use var):

In [40]:

```python
# will print 'pear' 5 times
['pear' for i in range(5)]
```

Out[40]:

```
['pear', 'pear', 'pear', 'pear', 'pear']
```

In [41]:

```python
# will print each value in uppercase
[s.upper() for s in ('apple', 'orange', 'peach')]
```

Out[41]:

```
['APPLE', 'ORANGE', 'PEACH']
```

You can also optionally include a condition on whether or not an element is created in the new list:

In [42]:

```python
fruits = ('apple', 'pear', 'orange', 'peach', 'cherry')
[f for f in fruits if len(f) > 5]
```

Out[42]:

```
['orange', 'cherry']
```

It can be especially useful to use a comprehension on nested sequences:

In [43]:

```python
# adds the pairs together
pairs = [(1,2), (3,4), (5,6)]
[x + y for x, y in pairs]
```

Out[43]:

```
[3, 7, 11]
```