Python Datatypes

Lists

- Ordered - once created, order of items does not change
- Comma-separated values in []
- Types can be mixed, but typically homogeneous
- Indexable by 0-based index
- Can use `+` to concatenate lists

In [1]:
```
years = [1620, 1215, 1812, 1941]
weird_list = [1, 'two', (3, 4), False]
print(years[1], weird_list[2])
both = years + weird_list
print(len(both))
print(both[-3])     # negative number indexes from end of list
```

```
1215 (3, 4)
8
two
```

List operations

To create empty list, use `[]` or `list()` (`[]` is more idomatic)

Operation	Description
append()	Used to add single element to end of list
extend()	Used to add multiple elements to end of list
insert()	Used to insert a single element at specific position in list
remove()	Used to remove element from list by value
pop()	Used to remove element from any position in list
reverse()	Used to reverse elements in a list

Operation	Description
len()	Returns length of list
min()	Returns minimum value (requires list of all same type)
max()	Returns maximum value (requires list of all same type)
count()	Returns number of occurrences an element value in list
index()	Returns position of first occurrence of element in list
sort()	Sorts list in ascending order
clear()	Empties list of all elements
copy()	Performs "shallow" copy of a list

In [2]:

```
fruits = ['banana', 'apple', 'orange', 'lime']
print(fruits)
fruits.append('lemon')
fruits.extend(['grape', 'strawberry'])
print(fruits)
fruits.insert(2, 'lime')
print(fruits)
fruits.remove('lime')
print(fruits)
# fruits.remove('papaya')
new_fruits = fruits
new_fruits[3] = 'mango'
print(fruits)
copy_fruits = fruits.copy()
copy_fruits[3] = 'dragonfruit'
print(fruits)
print(copy_fruits)
```

```
['banana', 'apple', 'orange', 'lime']
['banana', 'apple', 'orange', 'lime', 'lemon', 'grape', 'strawberry']
['banana', 'apple', 'lime', 'orange', 'lime', 'lemon', 'grape', 'strawberry']
['banana', 'apple', 'orange', 'lime', 'lemon', 'grape', 'strawberry']
['banana', 'apple', 'orange', 'mango', 'lemon', 'grape', 'strawberry']
['banana', 'apple', 'orange', 'mango', 'lemon', 'grape', 'strawberry']
['banana', 'apple', 'orange', 'dragonfruit', 'lemon', 'grape', 'strawberry']
```

Slicing operator

- Provides access to sub-section of list
- Uses `<list name>[s:e]` to specify start and end
- Returns element at start through element up to (but not including) end
- `[:]` used to access all items - both start and end are optional
- Can be used for both retrieve and update of list items
- Also works with strings

In [3]:
```
fruits = ['banana', 'apple', 'orange', 'lime']
print(fruits[1:4])
fruits[2:4] = ['mango', 'papaya', 'lemon']
print(fruits)
print(fruits[:3])
print(fruits[1:])
print(fruits[:])
print('Hello World'[3:5])
copy = fruits[:]        # optional mechanism to create shallow copy
copy[1] = 'grape'
print(fruits)
print(copy)
```

```
['apple', 'orange', 'lime']
['banana', 'apple', 'mango', 'papaya', 'lemon']
['banana', 'apple', 'mango']
['apple', 'mango', 'papaya', 'lemon']
['banana', 'apple', 'mango', 'papaya', 'lemon']
lo
['banana', 'apple', 'mango', 'papaya', 'lemon']
['banana', 'grape', 'mango', 'papaya', 'lemon']
```

Tuples

- Immutable
- Generally imply some structure
- One tuple generally describes one object (person, building, country, etc.)
- Parens not required when declaring

In [4]:
```
t = 'Gutzon Borglum', 'Idaho', 1867
print(t)
# t[1] = 'Montana'      NO - immutable
print(t[1])
```

```
location, state, year = t
print(location,state,year)
```

```
('Gutzon Borglum', 'Idaho', 1867)
Idaho
Gutzon Borglum Idaho 1867
```

In [5]:
```
# empty tuple
t = ()
print(t)
# singleton tuple
t = 1,
print(t)
```

```
()
(1,)
```

In [6]:
```
# use case for a singleton tuple: concatenation
t + (2,)
```

Out[6]:
```
(1, 2)
```

In [7]:
```
# another use case for singleton tuple:
# enables you to pass a single value to a function which takes an iterable
def func(iter):
    for item in iter:
        print(item, end=' ')
    print()

func('hello')
func((9,))
# func(9)
```

```
h e l l o
9
```

Sets

- unordered
- no duplicates

In [8]:
```
t = set()
type(t)
```

Out[8]:
```
set
```

In [9]:
```
even = { 2, 4, 6 }
print(even)
even.add(8)
even.add(2)
print(even)
```

```
{2, 4, 6}
{8, 2, 4, 6}
```

In [10]:
```
prime = set([int(x) for x in '2357'])
print(prime)
print('all numbers =', prime | even)
print('even primes =', prime & even)
```

```
{2, 3, 5, 7}
all numbers = {2, 3, 4, 5, 6, 7, 8}
even primes = {2}
```

- `add()` can be used to add a single item to set
- `update()` can be used to add multiple items to set
- `remove()` can be used to remove item; raises KeyError if absent
- `discard()` removes item if it exists

Dictionaries

- unordered list of key/value pairs
- associative array, hash, etc.

In [11]:
```
d = { 'red': 0, 'blue': 1, 'green': 2 }
d['blue'] = 9
d['yellow'] = -1
print(d)
```

```
{'red': 0, 'blue': 9, 'green': 2, 'yellow': -1}
```

In [12]:
```
d = {}
d['tall'] = 12
d['grande'] = 16
d['venti'] = 20
print(d)
```

```
{'tall': 12, 'grande': 16, 'venti': 20}
```

In [13]:
```
print('keys are', d.keys())
print('values are', d.values())
print('items are', d.items())
```

```
keys are dict_keys(['tall', 'grande', 'venti'])
values are dict_values([12, 16, 20])
items are dict_items([('tall', 12), ('grande', 16), ('venti', 20)])
```

In [14]:
```
keys = list(d.keys())
print(keys)
```

```
['tall', 'grande', 'venti']
```

In [15]:
```
d_keys = d.keys()
# now add to the dict...
d['trenta'] = 31
print(keys)
print(d_keys)
```

```
['tall', 'grande', 'venti']
dict_keys(['tall', 'grande', 'venti', 'trenta'])
```

- If element at requested index does not exist, `[]` access will cause error
- Can use `in` operator to check for existence before accessing
- Or use `get()` method to do both in single statement
- `del()` can be used to delete an element

In [16]:
```
# print(d['large'])
if 'large' in d:
    print(d['large'])
else:
    print('unknown')
```

```
print(d.get('large', 'default'))
print(d)
del d['trenta']
print(d)
```

```
unknown
default
{'tall': 12, 'grande': 16, 'venti': 20, 'trenta': 31}
{'tall': 12, 'grande': 16, 'venti': 20}
```

Exercise One

- Write a function called `minmaxavg` which accepts a float list and returns 3 values:
  - Minimum value in list
  - Maximum value in list
  - Average of all values in list
- Prompt the user for number of values
- In a loop, prompt the user for each float value and build into a list
- Pass list to `minmaxavg` and display results

Exercise Two

- Write a function called `fibonacci` that accepts an integer that is greater than 2
- This integer will represent the number of Fibonacci numbers the user is requesting
- Fibonacci numbers are: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- Each new number is the sum of the previous two
- Build a list of the requested set of numbers in the function, return from function, and print

Exercise Three

- Create a Python programmer that allows you to convert from Roman numeral to Arabic equivalent
- M: 1000, D: 500, C: 100, L: 50, X: 10, V: 5, I: 1
- MCLX = 1000 + 100 + 50 + 10 = 1160
- Prompt user for Roman numeral, convert to Arabic, and print result
- EXTRA: If you have time, deal with case where smaller number precedes a larger number (MCMLCIX = 1000 + (1000 - 100) + (100 - 50) + (10 - 1) = 1959

Exercise Four (If Time Permits)

- Create four calculator functions: addition, subtraction, multiplication, division
- Each function should take 2 integers and return an integer result
- Use a dictionary to map the strings representing the operators to the functions
- That is, "+" would be mapped to addition(), "-" would be mapped to subtraction()...
- Have your program read in lines like "2 + 4" and have it determine the result by parsing the line and then using the operator ("+" in this case) to find the appropriate function to invoke
- Call the function and output the result
- Hint: Functions are also objects in Python