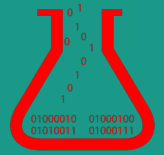
Variable Types



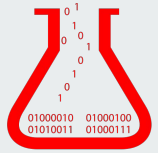
Python Collection Types

Agenda

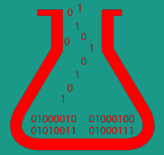
Lists
Dictionaries
Tuples
Sets
Indexing & Slicing
Mutability
Useful Built-In Functions

Python's Collection Types

- List
- Dictionary (dict)
- Tuple
- Set

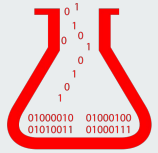


Lists



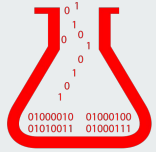
Lists

- A sequence of items that have unlimited length, known order, can be mixed data types, mutable
    - y = [1,2,3]
    - mylist = [“the”, “cat”, “in”, “the”, “hat”]
    - another_list = [1, “the”, 3.45, True]



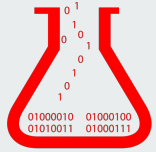
List - Defining & Operators

- x = list((1,2,3)) or x = [1,2,3]
  - Creates a list
  - To create a blank list use: x = list() or x = []
- mylist = [4,5,6]
- x + mylist → [1,2,3,4,5,6] (adds both lists together but doesn't assign it to anything)
- x * 3 → [1, 2, 3, 1, 2, 3, 1, 2, 3] (makes 3 copies of list but doesn't assign it to anything)



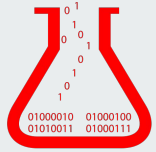
List - Methods

- x = [1,2,3]
- .append() - adds an item to the end of the list
  - x.append(4) → [1,2,3,4]
- .remove() - removes an item from the list (from the end)
  - x.remove(4) → [1,2,3]
- .pop() - pops an item of the end of the list and returns it
  - x.pop() → 3
  - x.pop(0) → 1 (pop(0) = front of the list)
- .extend() - extends the first list by adding the 2nd list to it
  - y = [9,10]
  - # x = [2]
  - x.extend(y) → [2,9,10]

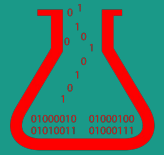


List - Joining a list together

- **str.join()**
  - Used to concatenate a sequence of strings into one string
  - .join() takes a list as argument
  - separator = “-”
  - sequence = [“join”, “me”, “together”]
  - separator.join(sequence) = “join-me-together”
  - “ ”.join(sequence) = “join me together



Dictionaries



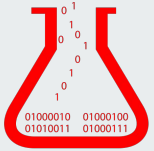
Dictionary (dict)

- A mutable unordered set of key:value pairs, with unique keys
- Syntax: sound_dict = {“cat”: “meow”, “duck”: “quack”}
- To access a given value, call the key:
    - sound_dict[“duck”] → “quack”
- To assign a new key:value pair or reassign an existing key:
    - sound_dict[“cow”] = “moo”
    - print(sound_dict) → {“cat”: “meow”, “duck”: “quack”, “cow”: “moo”}

Dictionaries

- Dictionaries are key:value pairs.
- Think of them like lists, but instead of numeric indexes (0,1,2,3,4…), the keys are arbitrary strings of text (or other hashable type).
- There are multiple syntaxes to make dictionaries
- Keys are unique and immutable (strings)*
- The dictionary is mutable (key:value pairs can be added or removed)

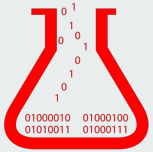
* Dict keys do not have to be strings, though the full definition of what can be used as a key is a bit beyond the scope of this class. Check out this page for more info: https://wiki.python.org/moin/DictionaryKeys



Dictionary instantiation

We can instantiate dictionaries with a few different syntaxes

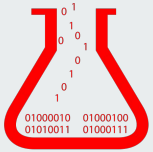
```
barn_animalweights = {'Cat':10, 'Dog':25, 'Elephant':2000, 'Giraffe':1000}
```



Dictionary instantiation

We can lookup values with bracket notation

```
barn_animalweights = {'Cat':10, 'Dog':25, 'Elephant':2000, 'Giraffe':1000}

print(barn_animalweights['Cat'])
```

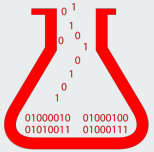


Dictionary instantiation

We can lookup values with bracket notation

```
barn_animalweights = {'Cat':10, 'Dog':25, 'Elephant':2000, 'Giraffe':1000}

print(barn_animalweights['Cat'])
```

```
...: print(barn_animalweights['Cat'])
10
```

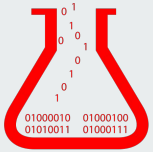


Dictionary instantiation

We can use bracket notation to modify, a dictionary

```
barn_animalweights = {'Cat':10, 'Dog':25, 'Elephant':2000, 'Giraffe':1000}
```

```
[10]: barn_animalweights['Dog'] = 45
  ...: print(barn_animalweights)
Cat': 10, 'Dog': 45, 'Elephant': 2000, 'Giraffe': 1000}
```

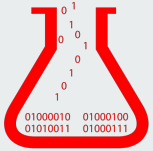


Dictionary instantiation

We can also use bracket notation to create a dictionary

```
barn_animalweights = {}
print(barn_animalweights)
#prints an empty dictionary
barn_animalweights['Cat']=10
barn_animalweights['Dog']=25
barn_animalweights['Elephant']=2000
barn_animalweights['Giraffe']=1000
print(barn_animalweights)
```

```
{}
{'Cat': 10, 'Dog': 25, 'Elephant': 2000, 'Giraffe': 1000}
```



Dictionary Comprehensions

Consider the for loop:

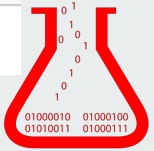
```
1  numbers = [1,2,3,4,5]
2  squares = [1,4,9,16,25]
3  square_dict = {}
4
5  for number,square in zip(numbers,squares):
6      square_dict[number] = square
7
8  print(square_dict)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Versus the comprehension:

```
1  {number:square for number,square in zip(numbers,squares)}
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```



Dictinoary Comprehensions

Consider the for loop:

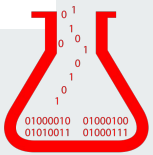
```
#Lets make a dictionary of animals and thier count on the farm

# consider the for loop
animal_Ls=['chicken', 'donkey', 'hippo']
count_ls=[10,100,50]
count_dict={}

for i in range(len(animal_Ls)):
    count_dict[animal_Ls[i]]=count_ls[i]

print(count_dict)
```

```
{'chicken': 10, 'donkey': 100, 'hippo': 50}
```

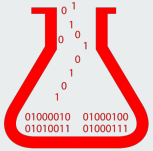
Versus the comprehension:

```
{k:v for k,v in zip(animal_list, count_ls)}
```

```
{'chicken': 10, 'donkey': 100, 'hippo': 50}
```



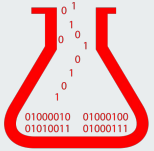
Dictionaries



Dictionaries

- Dictionaries are key:value pairs.
- Think of them like lists, but instead of numeric indexes (0,1,2,3,4…), there are arbitrary strings of text.
- There are multiple syntaxes to make dictionaries
- Keys are unique and immutable (strings)*
- The dictionary is mutable (key:value pairs can be added or removed)

- * Dict keys do not have to be strings, though the full definition of what can be used as a key is a bit beyond the scope of this class. Check out this page for more info: https://wiki.python.org/moin/DictionaryKeys

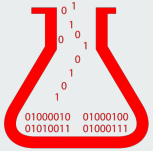


Dictionary exploration

- We made 3 dictionaries
- Consider the bound method ".update()" that can merge 2 dictionaries

```
1 print(farm_dict)
2 print(alt_farm_dict)
3 print(alt_farm_dict2)
```

```
{'donkey': 5, 'horse': 2, 'pig': 10}
{'hippo': 2, 'chicken': 200}
{'horse': 2, 'chicken': 47}
```



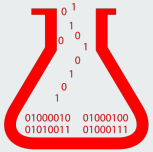
Dictionary mutation

- We change the dictionary by merging a second dictionary with it

```
2 # use the bound method update to change these in place
3 farm_dict.update(alt_farm_dict)
```

```
1 farm_dict
```

```
{'chicken': 200, 'donkey': 5, 'hippo': 2, 'horse': 2, 'pig': 10}
```

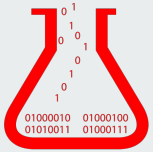


Dictionary exploration

- Get keys, values, or both back

```
4 print(farm_dict.keys())
5 print(farm_dict.values())
6 print(farm_dict.items())
7
```

```
dict_keys(['donkey', 'horse', 'pig', 'hippo', 'chicken'])
dict_values([5, 2, 10, 2, 200])
dict_items([('donkey', 5), ('horse', 2), ('pig', 10), ('hippo', 2), ('chicken', 200)])
```



Dictionary exploration

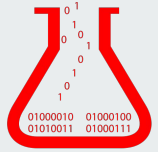
- Recall an item based on its keys this can be done with dictionary notation or using the “get” method

```
1 print("this uses the get method")
2
3 print(farm_dict.get('donkey'))
4
5 print("this is dictionary notation:")
6
7 print(farm_dict['donkey'])
8
```

```
this uses the get method
5
this is dictionary notation:
5
```

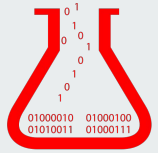
Dictionary Methods and Operations

- sound_dict = {“cat”: “meow”, “duck”: “quack”, “cow”: “moo”}
- .keys() - returns a list of the keys of the dictionary
  - sound_dict.keys() → dict_keys(['cat', 'duck', 'cow'])
- .values() - returns a list of the values of the dictionary
  - sound_dict.values() → dict_values(['meow', 'quack', 'moo'])
- .items() - returns a list of tuples of (key,value)
  - sound_dict.items() → dict_items([('cat', 'meow'), ('duck', 'quack'), ('cow', 'moo')])

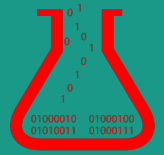


Advanced Dictionary Method - zip

- **zip()**
- Used to pair up the elements of two lists (or other iterable) based on shared index
  - odd = (1,3,5), even = (2,4,6)
  - print(list(zip(odd, even))) → [(1,2),(3,4),(5,6)]
- Can also be used with dictionaries:
  - students = [“Matt”, “Jane”, “Bob”], grades = [82, 97, 70]
  - print(dict(zip(names, grades))) → {“Matt”:82, “Jane”:97, “Bob”:70}

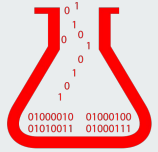


Tuple

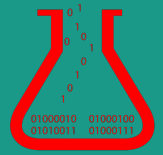


Tuple

- An immutable ordered list with a known number of elements.
  - Syntax: x = (1,4,6)
  - Immutability refers to the inability to be changed after the original assignment.
  - Tuples, are considered a primitive data type and like all the primitive data types, are immutable.

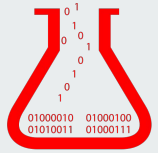


Set

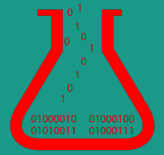


Set

- An unordered collection of UNIQUE items.
  - Syntax: x = {4,1,6} or x = set((4,1,6))
    - If y = {4,4,6,1} ⟶ y = {4,6,1} (the extra 4 is removed because its not unique item)
  - Cannot update an item only add or remove
    - set.add() ⟶ adds that item to the set
      - x.add(7) -> {1, 4, 6, 7}
    - remove() ⟶ removes that item from the set
      - x.remove(1) -> {6, 4, 7}

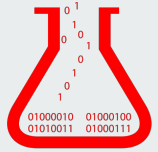


Indexing & Slicing



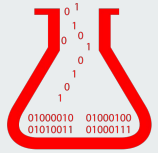
Indexing

- An iterable is any data type that can be used in a sequential fashion to find the next item, which includes string, list, tuple, dictionary, etc.
- We use the iterable property when searching through the various items to find a specific item, which is called indexing:
  - mylist = [“the”, “cat”, “in”, “the”, “hat”]
- Python is ‘zero-based’ so indexing for the first item:
  - mylist[0] → “the”



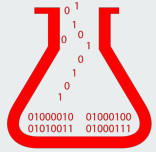
More Practice with Indexing

- mylist = [“the”, “cat”, “in”, “the”, “hat”]
  - mylist[1] ⟶ “cat”
  - mylist[-1] ⟶ “hat”
  - mylist[-4] ⟶ “cat”
- mystr = ‘python’
  - mystr[0] ⟶ ?
  - mystr[-1] ⟶ ?



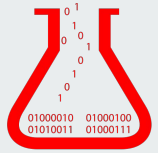
Slicing

- To call up a subset/part of a list, we use a slice
- Slice syntax = [# to start with, # to end on (does not include): step]:
  - If either of the first two numbers are left blank - defaults to the start or end of the iterable
  - If the step is left blank - defaults to a step of 1
- Examples: mylist = [“the”, “cat”, “in”, “the”, “hat”]
  - mylist[0:2] ⟶ [“the”, “cat”] (includes items 0 and 1, but not 2)
  - mylist[2:3] ⟶ [“in”] (only include item 2, equivalent to indexing mylist[2])
  - mylist[2:] ⟶ [“in”, “the”, “hat”] (the remainder of the list)
  - mylist[:-1] ⟶ [“the”, “cat”, “in”, “the”] (everything up to the last item)

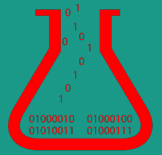


More Slicing Practice

- Examples: mylist = [“the”, “cat”, “in”, “the”, “hat”]
  - mylist[0:4:2] ⟶ ['the', 'in'] (first item then step of 2)
  - mylist[::-1] ⟶ ['hat', 'the', 'in', 'cat', 'the'] (reverses!)
  - mylist[4:8] ⟶ [‘hat’]
- Example: mystr = ‘Python’
  - mystr[0:2] ⟶ ?
  - mystr[4:6].upper() ⟶ ?
  - mystr[1:5:3] ⟶ ?
  - mystr[::-1] ⟶ ?

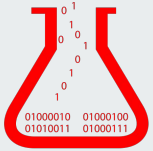


Mutability

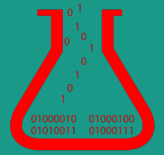


Mutability

- go to Jupyter notebook example
- (Lists.ipynb; section on copying lists)

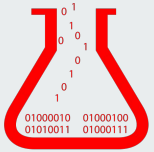


Useful Built-In Functions



zip()

- Used to pair up the elements of two lists (or other iterable) based on shared index
  - odd = (1,3,5), even = (2,4,6)
  - >> print(list(zip(odd, even)))
    - [(1,2),(3,4),(5,6)]
- Can also be used with dictionaries:
  - students = [“Matt”, “Jane”, “Bob”], grades = [82, 97, 70]
  - >> print(dict(zip(names, grades)))
    - {“Matt”:82, “Jane”:97, “Bob”:70}

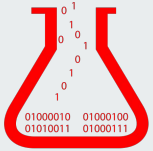


The sorted function

The sorted function makes a new list that is accessible when it is bound to an output variable

```
1  # since this is a series we can use methods that rely on order
2  farm_count=[200,  5, 2, 2, 10]
3
4  print (farm_count)
5
6  SortedLs=sorted(farm_count)
7
8  print (farm_count)
9
10 print (SortedLs)
```

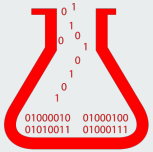
```
[200, 5, 2, 2, 10]
[200, 5, 2, 2, 10]
[2, 2, 5, 10, 200]
```



The .sort() bound methods

```
2
3 #the sort method changes the object itself
4 print ('the bound method changes the list itself')
5 farm_count.sort()
6 print(farm_count)
7
```

```
the bound method changes the list itself
[2, 2, 5, 10, 200]
```



Questions?

Contact:

Denis Vrdoljak
denis@bds.group