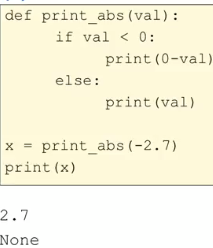
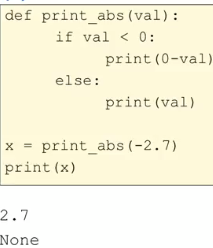
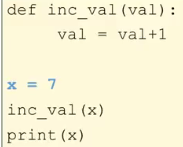
***Background in Python + Unix***

*Python*

* Jupyter 🡪 originally the combination of 3 languages, Julia, Python and R.
* These days, Jupyter supports over 40 programming languages.
* NumPy and Pandas libraries are incredibly valuable for doing data science in Python.
* Python is powerful + surprisingly fast for **interpretive language,** which aregenerally slower b/c they'd run on top of an **interpreter** rather than being compiled directly from the machine on which they're running.
* Think of this as having a middle man between the code + the machine, which isn't there for a language like C or C++.
* Python works well with other languages + is often used as a **blue language**, running in between components written in other languages.
* Python plays well with other languages in Jupyter notebooks as well.
* Python, because it's interpreted, can run everywhere w/ Python installed
* B/c Python is open source, you can install it anywhere w/out worrying about licenses.
* Python has a number of features, like **dynamic typing** (no explicit types needed for variable **declaration**) + automatic memory management, which make it both easy to learn + read
* Everything in Python is an **object**
* “x = 3” causes a PyIntObject to be created, which hold the value of the object, in this case 3, along w/ other details for Python to work w/ under the hood (including the type of object, # of references to the object, etc.)
* When you say x = 3, Python creates a PyIntObject with a value of 3 + have your variable x point directly to that object.
* x is created on the **stack** (holds local values + is managed by the program) + the PyIntObject is created on the **heap** (holds dynamically-created data + is managed by the OS)
* If we then say x = 4.5, The **garbage collector** in Python automatically frees the space associated w/ that 1st PyIntObject b/c nothing is pointed to it anymore, and now we have a PyFloatObject
* If ever curious to know if 2 variables are pointing to the same object, use the **is** command (False if x points to a PyIntObject and y points to a PyFloatObject)
* To test for *numeric* equality, use the **==** operator.
* x.lower() 🡪 lower **method**
* Strings CANNOT be changed 🡪 using x.lower on a string returns a new value, but the x variable still holds the original value
* To change it, re-declare x w/ the new value 🡪 x = x.lower() 🡪 x now points to new lowercase object
* If a function doesn't return anything, it effectively returns type None in Python.

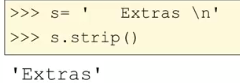
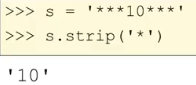
 



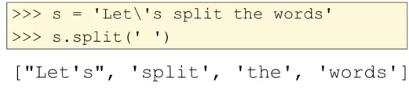
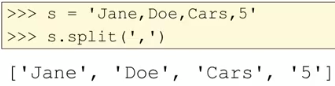
* This returns 7 b/c x is assigned the value 7 🡪 variable x is created + *points* to an PyIntObject w/ value 7
* When we call our function on the argument x, a *new* variable **val** is created, which points to the *same* PyIntObject as x (**copying the reference**) + they’re NOT the same variable
* Val is a *copy* of the pointer = points to same value as x
* When assigning val = val + 1, we are reassigning val to point to a new PyIntObject with value of 8, so *x has not changed at all + when we print x, we get the original value*
* When function inc\_val ends, out **temp variable** val is terminated, and only x is left, unchanged
* Never want a **global var** to have same name as a function **parameter**, though many places do not recommend using global variables *at all*

*String Functions*

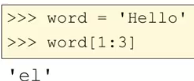
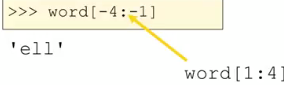
* Strings are **immutable** = cannot be changed 🡪 assign return values of calling methods on strings to new variables to save changes (or use In place = true for some)
* .**lower**(), .**upper**(), .**capitalize**() [1st char is capital]
* Concatenate w/ plus signs, replicate with asterisks
* **strip**(s[, chars]) 🡪 removes leading + trailing chats 🡪 omitting chats arg removes whitespace

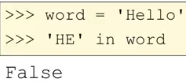
* Can **split** into smaller substrings

* **Slice** to get specific chars (w/ negative indices being equal to length of string – arg (5 – 4 = 1)

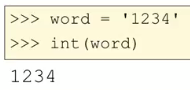
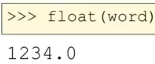
* Tell if substring is w/in string

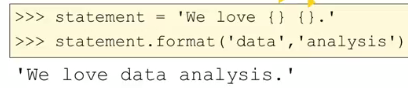
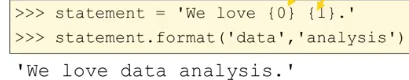
* Use find to find *where* substring is (get start indice)

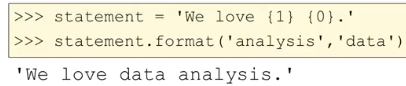


* Convert to numbers w/ int() and float(), but only if the string is a number

* Want to insert strings into larger string w/ format() + its placeholders



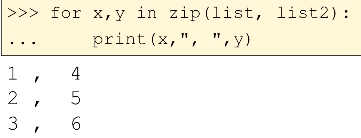
*Lists*

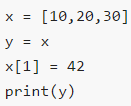
* Like arrays 🡪 resizable + has array implementation under the hood



* 1st loop is easier, more readable, less error-prone
* Are **mutable** 🡪 can be changed
* **.append()** 🡪 add elements to end of list
* **.pop(i)** 🡪 removes element at index i + returns it as well
* All remaining indices occurring after this index i shift to the left (become 1 less)
* Can remove specific values w/ **.remove(xx)**
* Can add 1 list to the end of another (merge) w/ **.extend(list2)f**
* Append 🡪 adds entire list as 1 element, extend 🡪 adds each list2 element as own element
* **.zip(list1,list2)** creates a paring of 2 lists 🡪 good for working w/ multiple lists at the same time



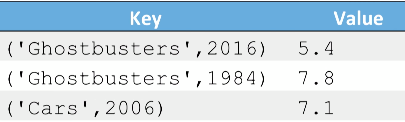
* For each x in list1 each y in list2, print x comma y
*  🡪 returns [10, 42, 30] b/c y points to same list object as x (not separate entities)
* i.e. we set up a list and made x point to it, then y points/refers to the *same list in memory­*
* x[1] = 42 changes the list in memory that both variables are pointing to
* To make y refer to its own copy of this list, do **y = list(x)**

*Tuples*

* **Tuples =** way of storing data in *immutable* manner 🡪 valuable for situations in data analysis
* Usually used to put together info that is in some way connected (like a record in a data frame)
* 
* **Notice multiple data types**
* Can index + loop over tuples like lists, but cannot change elements
* This is important b/c we can trust it to never change 🡪 Immutability is important in 2 key ways:
* Data which can change makes **parallel computing** (a key facet of data science) harder b/c we have to make sure everyone working on the problem sees those changes
* If data *can’t* change, we can just make copies of it to CPU nodes + no one has to worry about it changing on them while they’re computing
* Tuples can be used as the **keys** in **dictionaries**
* So the dictionary can organize based on the initial key value w/out worrying about a key being changed by someone w/ a reference to it data which can change

*Dictionaries*

* 1 of the most useful Python data structures 🡪 has **key-value (KV) pairs**
* Can ask if a key is in a factionary, and look up a value by its key (dictionaries are very fast at lookups)
* Only restriction on types 🡪 key must be immutable (value could be a list, or even another dictionary)
* *Any object is fine as a value in a KV pair, and everything in Python is an object*
* Can use a tuple as a key (b/c they’re immutable) to make unique keys

’



* Use keys as indices



* Can use **len()** to see how many KV pairs are in a dictionary
* To add to a dict, just write a new key as the index + assign it a value



* Would reassign this value if this key was already in the dict
* Dict’s are an **unordered collections** 🡪 no inherent ordering + we can’t trust its own internal ordering not to change at any time
* They’re very fast b/c they’re unordered
* If we need data ordered, dictionaries might not be ideal for the problem



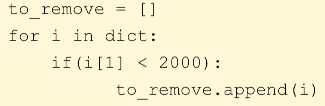
* Be careful using bracket access to get values b/c if the key doesn’t exist, we get an error
* 2 approaches mitigate this:
* Use **.get()** 🡪 if key doesn’t exist, get back value **None (test w/ x == None)**
* Test w/ **in 🡪** 
* Use these 2 methods to see if a key is in a dict before trying to access it
* Can return + remove items w/ **dict.pop(key)** 🡪 **mutates** the data structure such that the KV pair is no longer there
* Can also just use **del dict[(key)]** if we don’t want the value returned
* Can iterate over elements in dictionary
* Get all keys:



* Get all KV pairs:



* **.items()** returns a list w/ key and value
* *Cannot mutate a dict while iterating* 🡪 bad habit in any programming language to mutate a data structure while iterating through it b/c the underlying data structure implementation may change its structure of the data when adding/removing from it
* Ex: want to loop through a dict and remove everything meeting certain criteria 🡪 2 steps
* Find keys matching criteria via a loop 🡪 add that key to a list



* Remove items by iterating through the *list*, NOT the dict

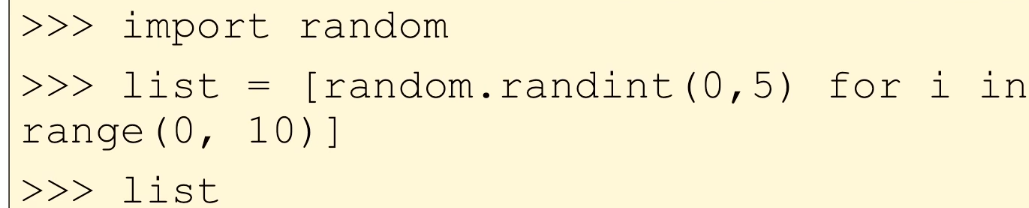


*Comprehension*

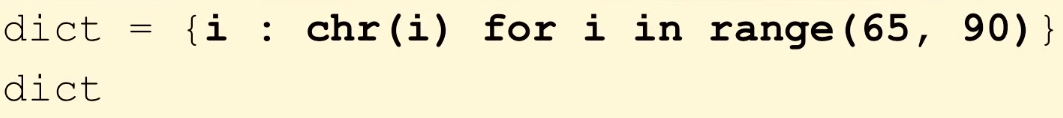
* Can builds lists + dictionaries easily w/ **comprehension**
* Ex: list of all squares from 1-10:

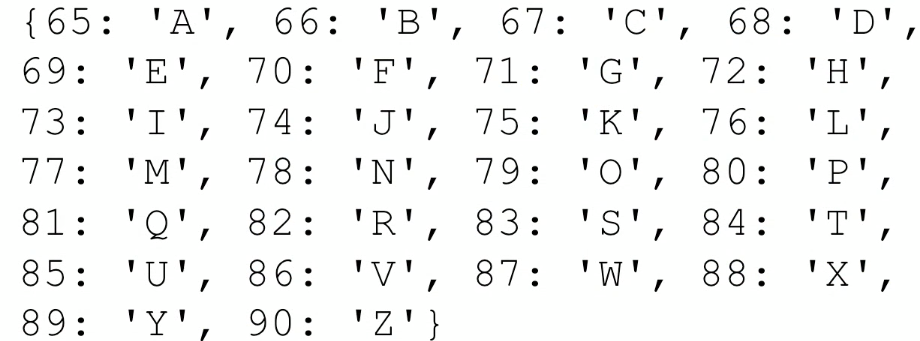


* Range(1,11) provides i w/ values 1-10, and we square each + store in the list



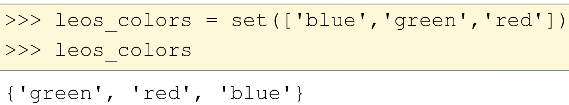
* Get a random int between 0-5 ten times
* Can build dictionaries as well, but while specifying both the key and the value
* Ex: ASCII chars w/ corresponding letters:



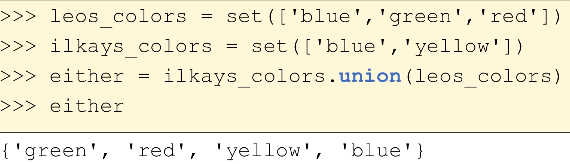


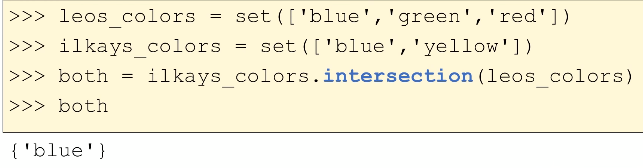
*Sets*

* **Sets =** support a # of useful math operations + only allow *unique* elements
* 3 useful qualities
* Unordered like dictionaries 🡪 allows them to be fast for a of key operations
* Hold unique elements 🡪 no dupes
* Support useful set operations like **UNION** and **INTERSECTION**
* **Pass lists into set() as an arg**



* Notice results are in a different order b/c sets are unordered
* Add to set w/ **set.add(‘value’**), so long as value is unique
* Remove w/ **.remove(‘value to remove’)** or **.discard(‘value to remove’)**
* Discard is better b/c if value isn’t there, it dose nothing, while .remove() causes an error
* **Set1.union(set2)** gives unique results out of both of the sets



* Useful to find ALL unique items in 2 separate sets
* Can also do this w/ **set 1 | set 2** 🡪 OR operator
* **Set1.intersection(set2)** finds all unique items that are in BOTH sets
* 
* Can also do this w/ **set 1 & set 2** 🡪 AND operator