Jupyter

Jupyter Notebook Keyboard Shortcuts

`a`	add cell above
`b`	add cell below
`dd`	delete cell
`enter`	edit cell
`shift + enter`	run cell
`esc`	escape editing
`y`	convert to code
`m`	convert to markdown
`shift + j`	select current and next cell
`shift + m`	merge selected cells
`ctrl + shift + minus`	split cells

Jupyter Notebook Markdown Syntax

`# text`	**Header 1**
`## text`	**Header 2**
`### text`	**Header 3**
	...
`**text**`	**bold**
`_text_`	*italics*
`~~text~~`	~~strikeout~~
`- text`	○ bullet list
`1. text`	1. ordered list
`` `text` ``	`inline code`
```` ```text``` ````	`code block`

Python Basics

Python Comments

`# comment`	Single-line or mid-line comment
`""" comment """` `''' comment '''`	Multi-line comment / docstring

Python Numeric Expressions

+	Addition
–	Subtraction/negation
*	Multiplication
/	Division
//	Division with integer outcome
%	Modulo/remainder
**	Power

The order of operations:

Parentheses-Exponents-Multiplication-Division-Addition-Subtraction

Python Primitive Data Types

`1, 2, 3,...`	Integer
`1.0, 2.0, 3.0,...`	Float
`'abc'`	String
`True, False`	Boolean

Recall:

- We use `str(x)`, `float(x)`, `int(x)` to change type
- We use `type(x)` to check an object's type

Python Comparisons

=	Assignment
==	Test of equality
!=	Test of inequality
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Python Strings

Python String Operations

`len(s)`	Get # characters in string
`s.lower()`	Convert to lowercase
`s.upper()`	Convert to uppercase
`s1 + s2`	Concatenate
`x in s`	Check if character(s) are in string
`s.count(x)`	Count occurrences of character(s)
`s.startswith(x)`	Checks if string starts with character(s)
`s.endswith(x)`	Checks if string ends with character(s)
`s.split(x)`	Split string on character(s)
`x.join(list)`	Join list of strings on character(s)
`s.replace(x1, x2)`	Replaces character(s) with other character(s)
`s.find(x, i)`	Returns position of character(s); (starts search at index 'i', if specified)

Python Special String Characters

`\`	Escape character
`\t`	Tab
`\n`	New line

Python Printing and Formatting

We've explored four approaches to printing:

`print("text", s)`	Separate items with commas
`print("text " + s)`	Concatenate items directly
`print(f"text {s}")`	Use implicit formatting
`print("text {s}".format(s='s'))`	Use explicit formatting

Python String Formatting

We format with syntax like:

```
{field_name:format_spec}
```

where `format_spec` takes the following form:

```
[[fill ]align][width][,][.precision][type]
```

character(s) to use for fill/padding

total # digits

comma-separated thousands

decimal digits

align	
<	left align
>	right align
^	center align

type	
f	float
%	percent
e	exponential

Python String Formatting

As an example, if we type

```
{x : y^z.wf}
```

We will format x to be a center-aligned (^) float (f) with w decimal digits of precision. The printed string will have z digits total, and any extra space will be filled with y.

Python String Indexing and Slicing

`s[i]`	Return character at position i
`s[-i]`	Return character at position i from the right
`s[i:]`	Return characters from i (inc) onwards
`s[:j]`	Return characters up to j (exc)
`s[i:j]`	Return characters from i (inc) to j (exc)

inc = inclusive, *exc* = exclusive

Recall:

- String indexing starts at position 0 when counting from the left.
- We can use the same indexing and slicing approach with lists/tuples.

Python Strings

Some other things to remember about strings:

- Strings are case sensitive
- We can create a multi-line string with blockquotes, e.g. : `"""string"""` or `'''string'''`
- We can use $>$ or $<$ to do string comparisons; order is determined by dictionary order, where:

  numbers $<$ uppercase letters $<$ lowercase letters

Python If-Else

Python Boolean Variables

Recall the rules for Boolean combinations:

```
True and True     =  True
True and False    =  False
False and False   =  False

True or True      =  True
True or False     =  True
False or False    =  False

not True          =  False
not False         =  True
```

Python if-else Statements

`if cond:`	Execute if condition is True
`elif cond:`	Execute if condition is True, and no preceding 'if' statement was executed
`else:`	Execute if no preceding 'if' statement was executed ("catch-all")

Recall:

- Indentation determines the lines affected by the 'if' statement
- We can nest 'if' statements

Python Data Structures

Python List Functions

`len(list)`	Get # elements in list
`sorted(list)`	Return sorted list
`max(list)`	Return maximum element
`min(list)`	Return minimum element
`sum(list)`	Sum all (numeric) list elements
`list.pop(i)`	Remove item at specified position**
`list.insert(i,x)`	Insert item at specified position**
`list.append(x)`	Append an item to list**
`list1.extend(list2)`	Append another list**
`list1 + list2`	Add two lists together
`x in list`	Check if item is in list
`list.index(x)`	Get the index of item
`list.count(x)`	Count appearances of item in list

** Modifies in place.

Python Complex Data Structures

`[1,2,3,...]`	List
`{1,2,3,...}`	Set
`(1,2,3,...)`	Tuple
`{1:'a', 2:'b', 3:'c', ...}`	Dictionary

Recall:

- Sets are unordered and have no duplicates
- Tuples are immutable; you cannot change their entries
- We can nest these data structures flexibly

Python Complex Data Structure Creation

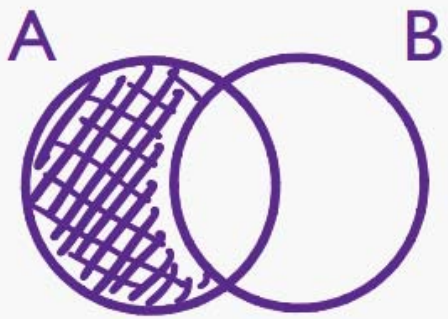
We can initialize empty data structures as follows:

`list()`	or `[]`	List
`set()`		Set
`tuple()`	or `()`	Tuple
`dict()`	or `{}`	Dictionary

Python Sets

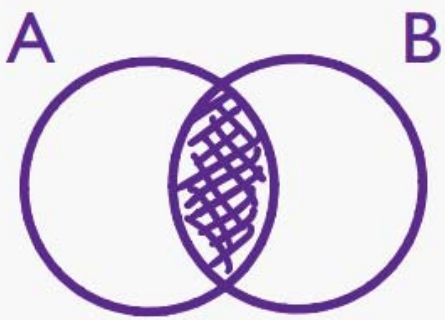
`len(set)`	Get number of items in set
`set.add(i)`	Add an item
`set.remove(i)`	Remove an item
`set_a - set_b`	Elements in a but not in b
	`set_a.difference(set_b)`
`set_a | set_b`	Elements in a and/or b
	`set_a.union(set_b)`
`set_a & set_b`	Elements in both a and b
	`set_a.intersection(set_b)`
`set_a ^ set_b`	Elements in a or b but not both
	`set_a.symmetric_difference(set_b)`
`set_a <= set_b`	Test if all elements in a are in b
	`set_a.issubset(set_b)`

Python Sets

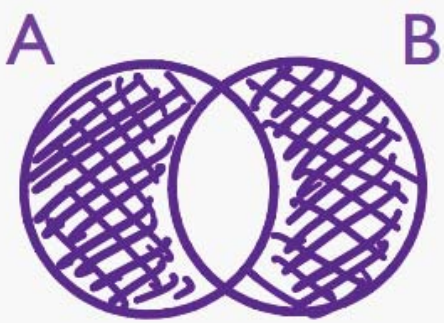


Difference: A - B

Union: A | B



Intersection: A & B



Symmetric Diff: A ^ B

Python Dictionaries

Recall that dictionaries have the structure:

```
dict = {key1 : val1, key2 : val2, ... }
```

We can access a specific value using its key:

```
dict[key1]
```

Similarly, we can assign a new value:

```
dict[key1] = val1_new
```

More generally:

`key in dict`	Check if key is in dictionary
`dict.pop(key)`	Remove key from dictionary
`dict.keys()`	Get a list of keys
`dict.values()`	Get a list of values
`dict.items()`	Get a list of (key,value) pairs

Python Loops

Python For and While Loops

We can create a while loop as follows:

```
while condition:
    do something as long as condition is met
```

We can create a for loop as follows:

```
for i in sequence:
    do something until no items left in sequence
```

The following tools are useful:

`break`	Exit the loop altogether
`continue`	Return to top of loop and continue
`range(start, stop, step)`	Create a list of numbers
`for k,v in dict.items():`	Iterate over a dictionary

Python Dictionary Updating

There are three main alterations you can make on a dictionary.

You can insert a new key:

```
dict[new_key] = new_val
```

You can overwrite an old key:

```
dict[old_key] = new_val
```

You can modify an old key, e.g.:

```
dict[old_key] = dict[old_key].append(new_val)
dict[old_key] += new_val
dict[old_key] = f(dict[old_key])
```

(function of old value)

Python Loops

There are a few main "flavors" of loops.

There are loops that count, e.g.:

```
i=0
while i<15:
    i+=1
```

There are loops that sum or extend, e.g.:

```
items=[]
for i in 'abcd':
    items.append(i)
```

Python Loops

Finally, are loops that aggregate or track, e.g.:

```
max_len = 0
max_item = 0
for i in ['a', 'bb', 'ccc']:
    if len(i)> max_len:
        max_len = len(i)
        max_item = i
```

Python Loops

We saw that loops can be shortened with list comprehensions:

```
for i in sequence:
    if condition:
        expression
```

$\longrightarrow$ `[ expression for i in sequence if condition ]`

Python Functions

Python Functions

We've seen many built-in functions:

- `print(x)`
- `max(x)`, `min(x)`, `sum(x)`, `len(x)`
- `int(x)`, `float(x)`, `str(x)` (change the type)
- `type(x)` (check the type)
- `range(start, stop, step)` (list # in range)
- `round(float, # digits)` (round a #)

Python Functions

We also saw some functions imported from packages.

`import math`	
`math.abs(x)`	Absolute value
`math.factorial(x)`	Factorial
`import random`	
`random.randint(from, to)`	Choose random integer in range
`random.random()`	Choose random float from 0 - 1
`random.choice(x)`	Choose random item from list/set/string
`import time`	
`time.sleep(seconds)`	Pause Python's execution of a program
`import string`	
`string.ascii_letters`	Returns all letters, 'abc...xyzABC...XYZ'

Python Functions

We can even write our own functions, using `def`:

```
def f_name(f_arguments):
    ... some code ...
    return f_result
```

Recall that:

- Arguments are variables or values that you pass into the function
  - These values are used within the function's code
  - They are optional; without them, the function takes no input
- Return statements are used to pass values out of the function
  - They return a result to the code that called your function
  - They are optional; without them, the function returns nothing

Python Functions

Some simple examples:

A function that takes no argument, and returns nothing:

i.e. just prints "hello"

```
def print_hello():
    print ('hello!')
```

A function that takes an argument, and returns a string:

i.e. `make_hello("world")` returns "hello! world"

```
def make_hello(name):
    return 'hello! ' + name
```