NIAPythonDay1

July 26, 2021

1 Welcome to the Data Analysis Workshop!

1.0.1 Copy course materials to your desktop from:

- BRC or VPN: `T:\TempTransfer\000_NIA_Python_Course`
- Otherwise: tinyurl.com/y2dnbsl2

1.1 Goals of this course

Students of this course should gain familiarity with: * Python's capabilities, strengths and weaknesses * Running Python code within a Jupyter Notebook environment * Generating tables/figures and doing simple stats * Keywords/names of concepts to help you Google things yourself

1.2 Target audience

- Bench scientists who want to analyze their own data
- Microsoft Excel users
- For beginners: No prior programming knowledge assumed

1.3 What will be covered

Day 1: Basic syntax, data types and operators

Day 2: Fancy data types (arrays and data frames)

Day 3: Exploratory data analysis

Day 4: Case study: Gene expression ribbon plot

Image Credit: Wolfram

1.4 About me

- Christopher Coletta, M.S.
- Computer vision, signal processing, machine learning, longitudinal data analysis
- Computer Scientist in Computational Biology & Genomics Core (CBGC)

1.5 About the Computational Biology & Genomics Core (CBGC)

- Core facility housed in LGG
- Room 10C222
- Seminar or training every month
- Two powerful Windows computers with lots of software and remote access available
- NIA IRP cloud computing like Amazon AWS via OpenStack
- NIAIRPGPU1 - server for deep learning (8x NVIDIA V100 GPUs)

1.6 What WILL be covered

1.6.1 Wrangle and Clean

- Simple and complex sort
- Filter
- Missing Data
- Group-by operations (Split/apply/combine)
- Merge two spreadsheets (JOIN operations)
- Bin continuous variables into categorical variables

1.6.2 Exploratory Data Analysis

- Summary statistics
- Pivot table
- Histogram
- Scatter plot
- Box and whiskers
- Pairwise scatterplot matrix

1.6.3 Model

- Linear regression

1.6.4 Visualize

- Facet plot
- Heatmap
- Manhattan plot

1.7 Why Python (vs. Excel, Matlab, R, etc)

- Free (no cost) and free (open-source)
- General-purpose: data, web, apps, microcontrollers
- Readible: simple, non-cluttered syntax
- Expressive: do more with less lines of code (relative to C)

- Popular: #3 language in TIOBE Index

1.7.1 Relative Strengths

Extensive, mature libraries for:

- Image analysis
- Video analysis (self driving cars platform)
- Machine learning, especially artificial neural networks
- Natural language processing (sentiment analysis)

1.7.2 Relative Weaknesses

- Mobile apps

1.8 Python Learning Resources

- SoloLearn phone app: Python 3 tutorial
- Python for Scientists and Engineers - Free Book by Shantnu Tiwari
- PyData YouTube channel
- Questions and answers on StackOverflow.com
- r/LearnPython on Reddit
- Use Jupyter built-in operator ?

1.9 Ecosystem of Python Data Analysis Software

Anaconda is one of many Python "distributions" that bundles core Python, essential 3rd party packages, and various IDEs.

2 Integrated Development Environment

The software app you use to build and test your code.

2.1 Python IDE Choices

- Spyder
- PyCharm
- Jupyter - IDE for this workshop
- And more …

2.2 Jupyter

- IDE for Python, R, Bash, many others
- Web browser interface: communicates with either local or remote back-end ("kernel")
- Creates a sharable document called a notebook.
- Notebook divided up into cells that contain code, output and documentation ("Markdown cell").

2.3 Jupyter Markdown Cell

- Markdown: Document-formatting style that is easly convertable to HTML
- Headings preceeded by #
- unordered lists preceeded by a *
- ordered lists preceeded by a number
- Math equations go in between two Dollar signs, example: $t = \frac{\hat{\beta} - \beta_{H_0}}{s.e.(\hat{\beta})}$
- Create links like this

2.4 Code Cells

- Python code goes in here
- Shift+Enter to run and goto the next cell
- Ctrl+Enter to run code and stay on current cell
- Upon execution, a number shows up on the left indicating order of execution.
- You don't have to run code cells in the order they appear in the notebook.

[1]:
```
print( "Hello, world!" )
```

```
Hello, world!
```

2.5 Interacting with cells: Command mode

- Press Esc - box turns blue
- Useful shortcuts:
  - b = Insert cell below
  - a = insert cell above
  - dd = Delete cell
  - Shift + up or down = select/highlight two or more cells
  - M = merge highlightes cells into one

2.6 Edit mode

- Double click to edit - box turns green
- Useful shortcuts
  - Ctrl + Shift + - = split cell at cursor location
  - Enter = gives you a new line inside the same cell
  - Shift + Enter = Runs the code in this cell and go to the next one
  - Ctrl + Enter = Runs the code in this cell and stay on this one

3 Basic Python Syntax

3.1 Comments

Lines preceededed by a hash symbol "#" are ignored by the Python interpreter

[2]:
```
# Run me! nothing happens!!!
# askfdjhdsakfadhsfadsk
print("before the hash") # after the hash
```

before the hash

3.2 Assignment, i.e., give a value a name

- An assignment is the name on the left side of an equal sign.
- It gives a name to a value.
- Names can have upper and lowercase letters, numbers (as long as it's not the first character), as well as underscores (Shift + -).
- Don't use a name that is also a Python Syntax keyword
- Assignment statements in Python do not copy objects, they create bindings between a target and an object.

[3]:
```
a_value = 42
```

See the value attached to the name by typing the name

[4]:
```
a_value
```

[4]: 42

3.3 print() function

Use the print function to output one or more values at once.

[5]:
```
print( a_value )
```

42

3.4 Code-completion using TAB key

Hit the TAB key to use code completion to help you type faster. Most IDEs have this option. Usually a pop-up menu will appear

[6]:
```
a_value
```

[6]: 42

3.5 Python Data Types: what are they, and why do we care?

- Different types of data, different data types
- Each type has their own various "superpowers," i.e., functionality.
- Advanced programmers often define their own types with their own functionality
- Here, "simple" means that these are types that are built into core Python, and you can use them right away.
- "Fancy" means simply that you need to use the import command before you use them.

3.5.1 Scalar Data Types (simple)

- integer int: counting numbers
- float float: decimal numbers
- boolean bool: true/false

3.5.2 Iterable Data Types (simple)

- string str: words
- list list: collection of things (ordered)
- dictionary dict: map one value to another (unordered)
- set set: unique collection of things (unordered)

3.5.3 What the difference between "scalar" and "iterable"?

- You can't loop over a scalar.

3.5.4 Iterable Data Types (fancy)

- NumPy multi-dimensional array: data, images
- Pandas DataFrame: spreadsheet analog

And many more…

3.6 Scalar Data Types: Integer (int)

- A counting number 1, 2, 3, -89 …, 0

[7]:
```
-23
```

[7]:
```
-23
```

[8]:
```
type( 2345 )
```

[8]:
```
int
```

[9]:
```
type( a_value )
```

[9]:
```
int
```

3.7 Scalar Data Types: Float (float)

- Decimal numbers
- An accurate approximation to many many decimal places, but technically not an EXACT representation
- If you want to know more about why decimal numbers are called "floats", click here.

[10]:
```
type( 3.14159 )
```

[10]:
```
float
```

[11]:
```
type( 1/3 )
```

[11]:
```
float
```

3.8 PEMDAS operators

1. Parentheses - ()
2. Exponent - **
3. Multiplication - *
4. Division - /
5. Addition - +
6. Subtraction - -

Example: What is $9 - 3 \div \frac{1}{3} + 1 =$?

[12]:
```
9 - 3 / 1/3 + 1
```

[12]:
```
9.0
```

[13]:
```
9 - 3 / (1/3) + 1
```

[13]:
```
1.0
```

3.9 Using the type() function

Use this to have Python tell you the data type of any expression or named value.

[14]:
```
type( a_value )
```

[14]:
```
int
```

[15]:
```
type( 3.14159 )
```

[15]:
```
float
```

3.10 Scalar Data Types: Boolean (bool)

Bools can only have a value of True or False.

[16]:
```
True
```

[16]:
```
True
```

[17]:
```
False
```

[17]:
```
False
```

[18]:
```
type( True )
```

[18]:
```
bool
```

3.11 Boolean operators and, or, and not

- and and or are "binary operators", meaning you slap them in between two truth values to make one value.

- Expression is evaluated left-to-right

[19]:
```
False and False
```

[19]: False

[20]:
```
True and True
```

[20]: True

[21]:
```
True or False
```

[21]: True

[22]:
```
False or True
```

[22]: True

[23]:
```
True or True
```

[23]: True

[24]:
```
False or False
```

[24]: False

[25]:
```
my_bool_value = True and False
print( my_bool_value )
```

```
False
```

not is a unary operator that negates the value after it.

[26]:
```
not True
```

[26]: False

A computer science subtlety: The short circuit 'or' operator

[27]:
```
True or False and False
```

[27]: True

[28]:
```
True or False and False # True
```

[28]: True

3.12 Some math operators

- < less than
- <= less than or equal to

- `>` greater than
- `>=` greater than or equal to
- `==` is equal to
- `!=` is not equal to

Note the double equal signs is an operator, not an assignment!!

[29]:
```
5 < 6
```

[29]:
```
True
```

[30]:
```
6 <= 6
```

[30]:
```
True
```

[31]:
```
-6 <= 6
```

[31]:
```
True
```

[32]:
```
6 != 6
```

[32]:
```
False
```

[33]:
```
True == True
```

[33]:
```
True
```

[34]:
```
not (True == True)
```

[34]:
```
False
```

3.13 Using whos command to keep track of named values

[35]:
```
whos
```

```
Variable        Type    Data/Info
---------------------------------
a_value         int     42
my_bool_value   bool    False
```

3.14 Iterable Data Types: Strings (str)

- A data type that contains one or more characters
- Strings are surrounded, a.k.a. "delimited" by matching single or double quotes
- You choose whether to use single or double quotes based on what's in the string.
- Escape characters: Backslash followed by a letter to render special characters
  - `\n`: New line
  - `\t`: Tab
  - `\"`: Quote character (not end of string)

[36]:
```
"Hello, world!"
```

[36]:
```
'Hello, world!'
```

[37]:
```
'Hello, world!'
```

[37]:
```
'Hello, world!'
```

I repeat: ***No difference between single and double quotes strings!!!!*** I promise!

[38]:
```
"Can't"
```

[38]:
```
"Can't"
```

[39]:
```
'"Really," she said?'
```

[39]:
```
'"Really," she said?'
```

[40]:
```
"I said, \"Hi my name is Chris\""
```

[40]:
```
'I said, "Hi my name is Chris"'
```

[41]:
```
" First line\n Second line"
```

[41]:
```
' First line\n Second line'
```

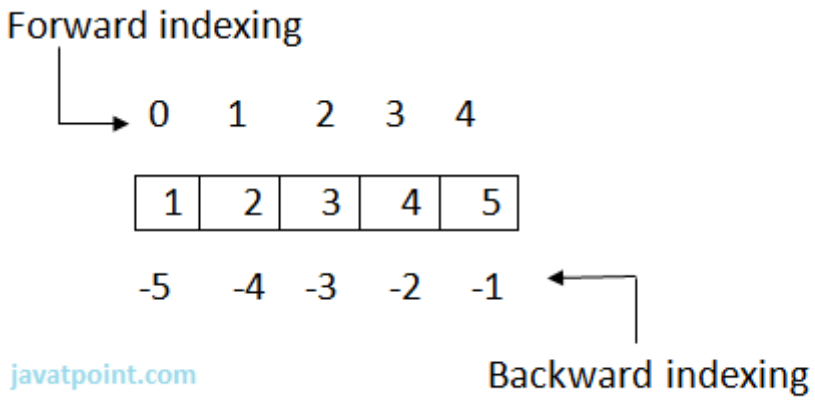
[42]:
```
print( " First line\n Second line" )
```

```
First line
Second line
```

By the way, I'm ***not*** talking about the backtick ‘, which shares a key with the tilde ~ character. Backtick is ***different*** than a single quote ’, which shares a key with the double quote ”.

3.15 Iterable Data Types: Lists (list)

- Container for a collection of values
- Can all be the same type or different, doesn't matter.
- Items delimited by commas, all surrounded by brackets [], not parentheses ()
- The order of the values in the list is remembered



[43]:
```
a_list = [ 1, 2, 3, 1, "a dog", 'a cat' ]
```

3.15.1 Get the ith element from a list using bracket notation

[44]:
```
a_list[0]
```

[44]: 1

[45]:
```
a_list[4]
```

[45]: 'a dog'

3.15.2 Negative index counts from the back of the list

[46]:
```
a_list[-1]
```

[46]: 'a cat'

3.15.3 Get position of a value within a list using .index()

[47]:
```
a_list.index('a cat')
```

[47]: 5

3.15.4 Use the "unpacking" syntax to get values out of small lists

[48]:
```
a_few_things = [ "hello", "goodbye", 42 ]
```

[49]:
```
a_few_things
```

[49]: ['hello', 'goodbye', 42]

[50]:
```
first, second, third = a_few_things
```

[51]:
```
first
```

[51]:
```
'hello'
```

[52]:
```
second
```

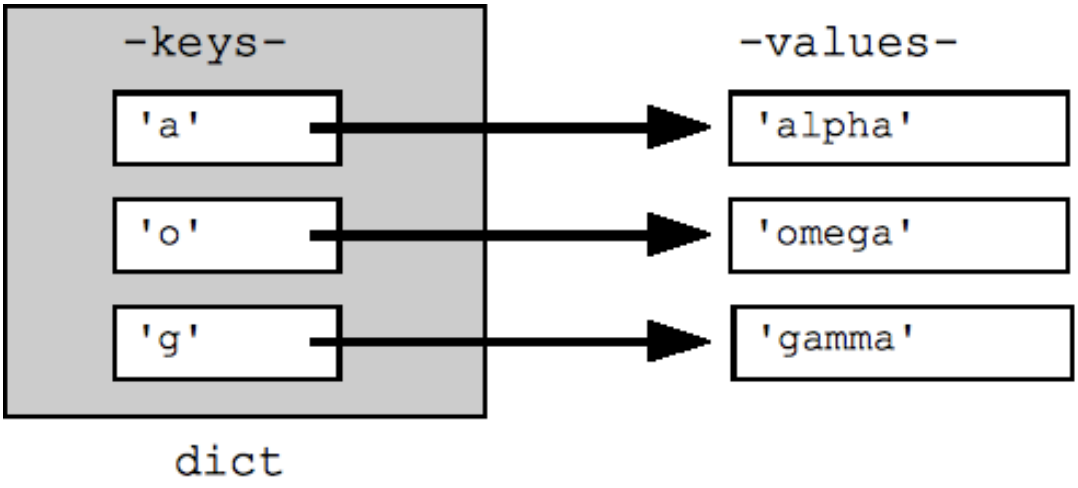
[52]:
```
'goodbye'
```

[53]:
```
third
```

[53]:
```
42
```

3.16 Iterable Data Types: Dictionaries (dict)

- A dict is one-way associative array, where "keys" are mapped to "values."
- Note: A dict does not keep track of the order in which you inputted the key-value pairs
  - for that you need collections.OrderedDict



3.16.1 Create a dict with stuff in it

The keys are separated by the values by a colon (:), and the key-value pairs are separated by commas.

[54]:
```
toy_dict = { 1 : 'a', 2 : 'b', 3: 'c'}
```

[55]:
```
toy_dict
```

[55]:
```
{1: 'a', 2: 'b', 3: 'c'}
```

3.16.2 Access an element in a dict using its key and bracket notation []

[56]:
```
info = { 'first name': "Chris",
         "last name" : "Coletta"}
```

[57]:
```
info
```

[57]:
```
{'first name': 'Chris', 'last name': 'Coletta'}
```

[58]:
```
info['first name']
```

[58]:
```
'Chris'
```

3.16.3 Keys, not values go into the dict, or you get an error

[59]:
```
info['Chris']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-59-34f6e1331251> in <module>
----> 1 info['Chris']

KeyError: 'Chris'
```

3.16.4 Create an empty dict

Declare empty dict with {}, or dict().

[60]:
```
type( {} )
```

[60]:
```
dict
```

3.16.5 Add a new key-value pair to an existing dict using bracket notation []

[61]:
```
toy_dict['new_key'] = 'new_value'
```

[62]:
```
toy_dict
```

[62]:
```
{1: 'a', 2: 'b', 3: 'c', 'new_key': 'new_value'}
```

[63]:
```
toy_dict = {}
```

[64]:
```
toy_dict
```

[64]:
```
{}
```

[65]:
```
toy_dict[1] = 'a'
toy_dict[2] = 'b'
toy_dict[3] = 'c'
toy_dict['new_key'] = 'new_value'
```

[66]:
```
toy_dict
```

[66]: {1: 'a', 2: 'b', 3: 'c', 'new_key': 'new_value'}

3.16.6 Get just the keys or just the values

Every dict has the built-in functions ("methods" in Pythonic speak) .keys() and .values()

[67]:
```
toy_dict.keys()
```

[67]: dict_keys([1, 2, 3, 'new_key'])

[68]:
```
toy_dict.values()
```

[68]: dict_values(['a', 'b', 'c', 'new_value'])

[69]:
```
toy_dict
```

[69]: {1: 'a', 2: 'b', 3: 'c', 'new_key': 'new_value'}

3.16.7 Values can be any other type, including iterables

[70]:
```
{ "former_value": ['a', 'b', 'c' ] }
```

[70]: {'former_value': ['a', 'b', 'c']}

3.17 Iterable Data Types: Sets (set)

- Similar to math concept of sets; has operations like union, intersection, etc.
- Sets are unindexed, unordered, and contains no duplicates.
- My personal favorite of the Python standard types!

3.17.1 Create a set with stuff in it

Declare a set by putting values inside braces.

[71]:
```
set('GATTACA')
```

[71]: {'A', 'C', 'G', 'T'}

[72]:
```
a_set = {'set', 'of', 'words', 'of'}
```

[73]:
```
a_set
```

[73]: {'of', 'set', 'words'}

3.17.2 Create an empty set

Make an empty using set().

[74]:
```
empty_set = set() # not {}, that would be an empty dict
```

[75]:
```
empty_set
```

[75]:
```
set()
```

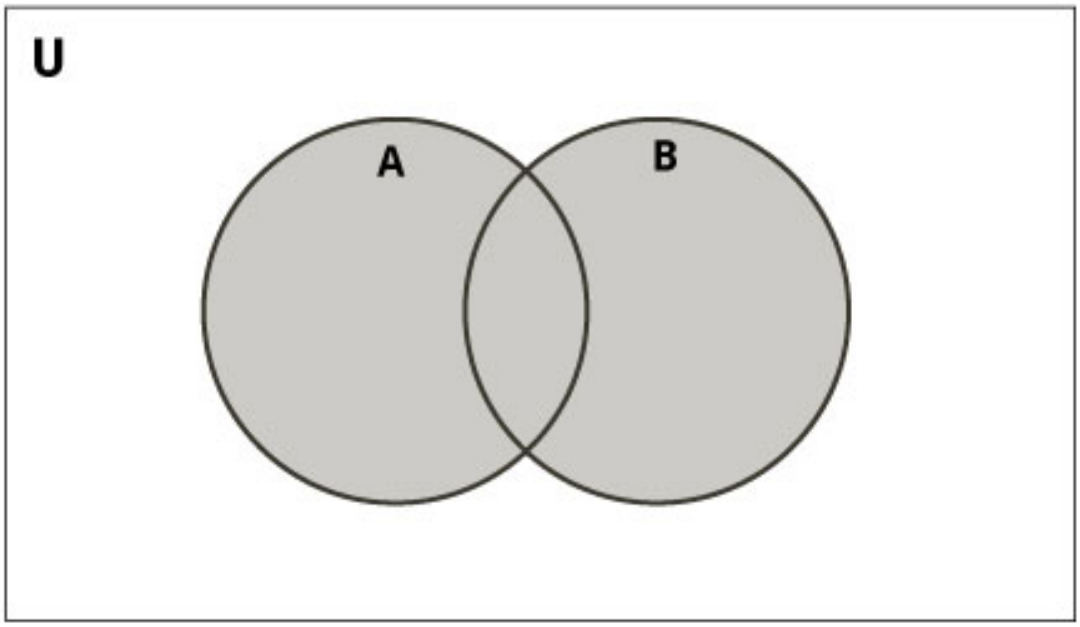
[76]:
```
empty_set.add( 'hi' )
```

[77]:
```
empty_set
```

[77]:
```
{'hi'}
```

[78]:
```
first = {1,2,3,4,5}
second = {4,5,6,7,8}
```

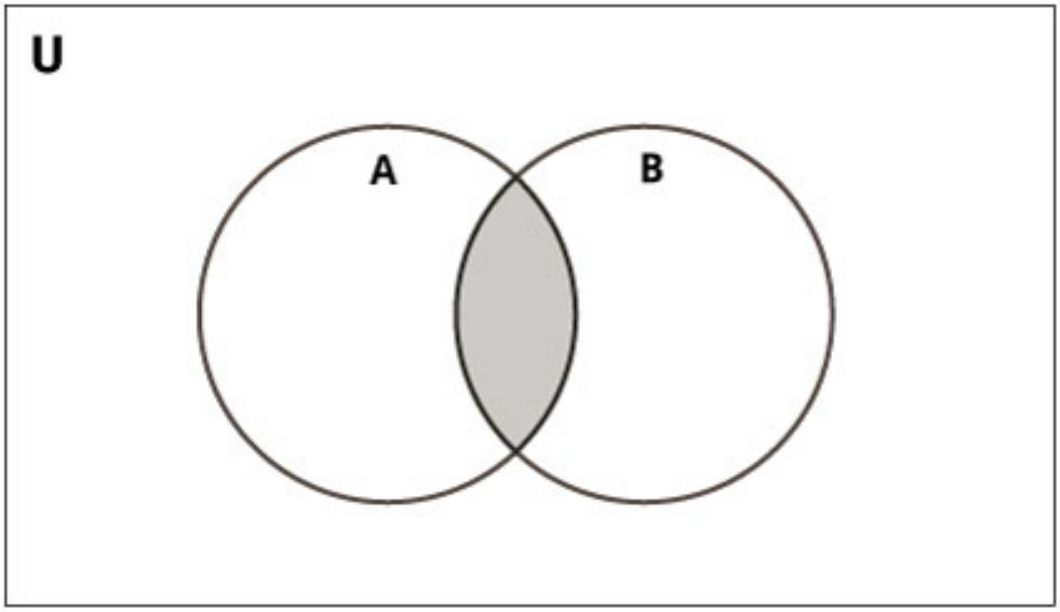
3.17.3 Set Union Operator (|) - "or"



[79]:
```
first | second
```

[79]:
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

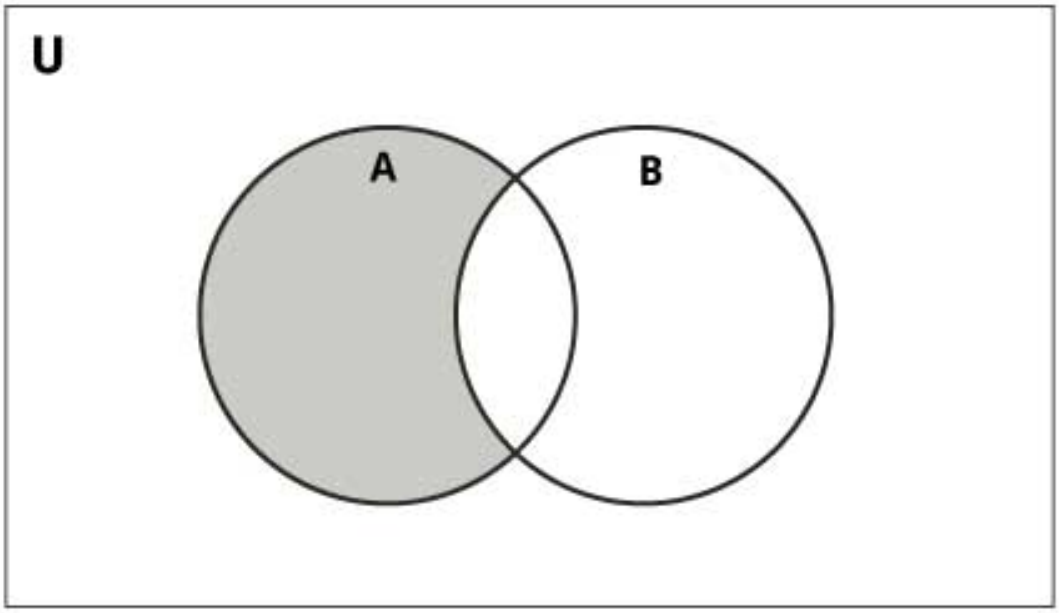
3.17.4 Set Intersection Operator (&) - “and”



[80]:
```
first & second
```

[80]:
```
{4, 5}
```

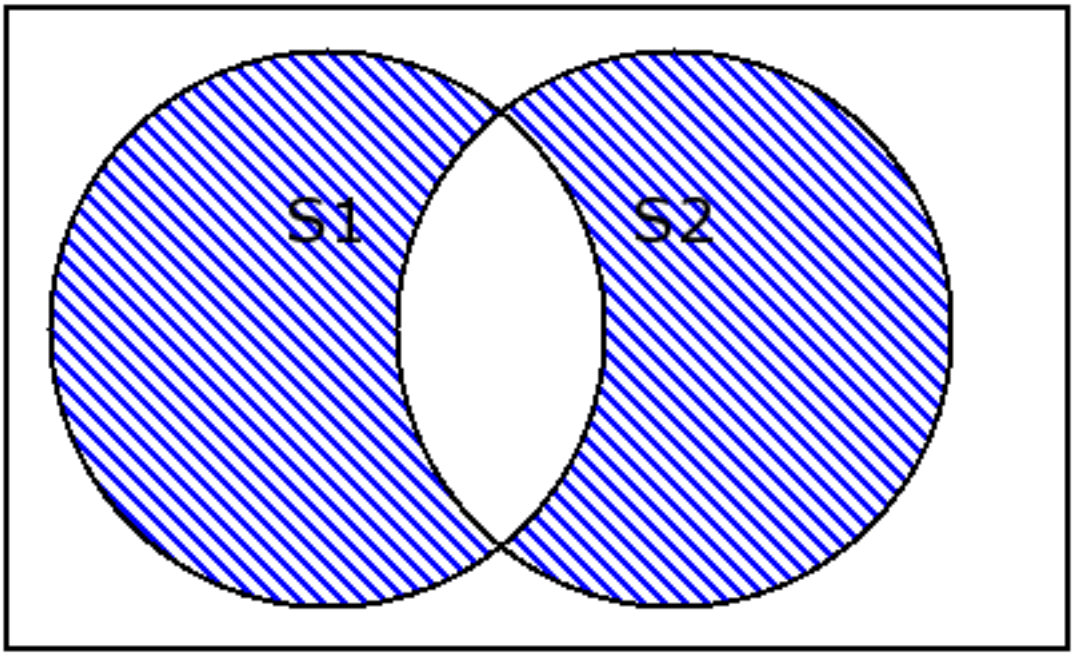
3.17.5 Set Difference Operator (-)



[81]:
```
first - second
```

[81]: {1, 2, 3}

3.17.6 Set Symmetrical Difference Operator (^)



[82]:
```
first ^ second
```

[82]: {1, 2, 3, 6, 7, 8}

3.18 How many elements in an iterable? Use len()

[83]:
```
len( first ^ second )
```

[83]: 6

[84]:
```
a_list
```

[84]: [1, 2, 3, 1, 'a dog', 'a cat']

[85]:
```
len(a_list)
```

[85]: 6

3.19 Can you change a value's type? Yes!

Use these functions to "coerce" a value from one type to another:

- `int()`
- `float()`
- `bool()`

- list()
- dict()
- set()
- et al.

[86]:
```
type( 45 )
```

[86]:
```
int
```

[87]:
```
type( '45' )
```

[87]:
```
str
```

[88]:
```
a_string = '45'
```

[89]:
```
56 + a_string
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-89-e1929fed3ecc> in <module>
----> 1 56 + a_string

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

[90]:
```
a_string
```

[90]:
```
'45'
```

[91]:
```
int( a_string )
```

[91]:
```
45
```

[92]:
```
56 + int(a_string)
```

[92]:
```
101
```

[93]:
```
float( '-45.0345' )
```

[93]:
```
-45.0345
```

[94]:
```
int( float( '-45.0345' ) )
```

[94]:
```
-45
```

[95]:
```
float( 45 )
```

[95]:
```
45.0
```

[96]:
```
str( 56 ) + a_string
```

[96]:
```
'5645'
```

[97]:
```
list( "listify me!" )
```

[97]:
```
['l', 'i', 's', 't', 'i', 'f', 'y', ' ', 'm', 'e', '!']
```

[98]:
```
round( 9.9 )
```

[98]:
```
10
```

[99]:
```
round( -9.9 )
```

[99]:
```
-10
```

[100]:
```
int( round( 9.9 ) )
```

[100]:
```
10
```

[101]:
```
set( "listify me!" )
```

[101]:
```
{' ', '!', 'e', 'f', 'i', 'l', 'm', 's', 't', 'y'}
```

[102]:
```
float( 3 )
```

[102]:
```
3.0
```

[103]:
```
int( 3.14159 )
```

[103]:
```
3
```

[104]:
```
bool( "a_string" )
```

[104]:
```
True
```

[105]:
```
bool( "" )
```

[105]:
```
False
```

[106]:
```
bool(  )
```

[106]:
```
False
```

3.20 Iterating over items in a list using a for loop

- Statements you want to be repeated inside the loop should be *indented* below the first line.
- Use the TAB key to indent.

- In between the for keyword and the in is the placeholder name whose value changes each time through the loop.

[107]:
```
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June']
```

[108]:
```
print( "before the loop" )

for m in months:
    print( m )


print( "after the loop" )
```

```
before the loop
Jan
Feb
Mar
Apr
May
June
after the loop
```

3.20.1 FYI: your temporary "placeholder" variable remains after the for loop

[109]:
```
m
```

[109]:
```
'June'
```

[110]:
```
del m
```

[111]:
```
m
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-111-9a40b379906c> in <module>
----> 1 m

NameError: name 'm' is not defined
```

3.21 Iterating over items in a dict using a for loop using .items() syntax

Use the unpacking syntax within the for .. in syntax to directly assign names to the key and value separately.

[112]:
```
num_days_in_month = { 'Jan' : 31,
                      'Feb' : 28,
                      'Mar' : 31,
```

```
                        'Apr' : 30 }
```

[113]:
```
num_days_in_month
```

[113]:
```
{'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30}
```

[114]:
```
num_days_in_month.items()
```

[114]:
```
dict_items([('Jan', 31), ('Feb', 28), ('Mar', 31), ('Apr', 30)])
```

[115]:
```
for m, d in num_days_in_month.items():
    print( "There are", d, "days in", m )
```

```
There are 31 days in Jan
There are 28 days in Feb
There are 31 days in Mar
There are 30 days in Apr
```

3.21.1 Without using .items() iterating over a dict will give you just the keys

[116]:
```
for thing in num_days_in_month:
    print( thing )
```

```
Jan
Feb
Mar
Apr
```

3.21.2 Advanced: Use a "dict comprehension" to switch directionality from value to key

[117]:
```
{ value: key for key, value in num_days_in_month.items() }
```

[117]:
```
{31: 'Mar', 28: 'Feb', 30: 'Apr'}
```

3.22 Day 1 review

1. Python ecosystem of tools
2. Jupyter Notebook is code, output and documentation all in one document
3. Type code into cells, and to run them you press Shift-Enter
4. Tab completion is nice
5. Different data types for different data
6. Operators take one or more input values and turn them into other values *based on the input values type*
7. Converting data from one type to another using the function syntax, e.g., int()
8. Iterating over iterables using a for loop