COMP1005 Python Cheat Sheet

Lisa Luff

8/12/2020

Contents

Python Version

- The command *python* will tell you which version you have, make sure to enter quit() after it put you in interpreter mode

Getting Started

Command	Purpose
python 3 *filename*.py	Runs a program in python, via Vim
#	To write a comment

Style

- Need to write a README for each practical, test and exam
- Comment at the start of each program:

```
'''
Author:
Student Number:

filename.py: Description of program

Modules: modulename - What it does

Revisions: xx/xx/xxxx - Changes made
'''
```

Flow of Execution

1. Import statements
2. Function definitions
3. Set up variables
4. Input data
5. Process data
6. Output data

Data Types

Command	Purpose
n	Is an integer (numeric)
n.n	Is a float (numeric)
*n*j	Is a complex number with j as the imaginary (numeric)
'*n*' or "*n*"	Is a string (non-numeric)
False, 0, 0.0, 0j, ' ', "", (), [], {} or None	Boolean False (non-numeric)
True, or anything other than explicitly Boolean False	Boolean True (non-numeric)

Variables

Command	Purpose
=	Assigns a variable
variable1, variable2 = *input1, input2*	Assigns mulitple variables at the same time
type(*variable*)	Outputs the type of data in the variable

Creating a Vector

- If numeric

```
variable = (x, y, z)
```

- Where x, y, and z are numeric values
- If non-numeric
  - Strings
    - An individual string can be considered a vector of characters

```
variable = "Type whatever"
```

```
* But you can also have a vector of multiple character strings
```

```
variable = ("These", "are", "strings")
```

Vector Operations

Command	Purpose
$x + y$	Concatenates x and y
$x * n$	x repeated n times

Indexing

- Starts at 0
- If you have x elements, the last element will be at x - 1, because the indexing started at 0
- Think of indexing values as being between the elements

Command	Purpose
variable[x]	Element x of *variable*
variable[-x]	Element x counting backwards from the last element of *variable*

Slicing

Command	Purpose
list = *variable*.split('x')	Turns *variable* into a list, separating the variable anywhere there is x
variable[*start*:*stop*:*step*]	Create a list(or array) of the elements from *variable* specified by the range of *start*:*stop*:*step* (works the same as range())

Basic Functions

Command	Purpose
print()	Echo's the output
round(x, y)	Rounds the value for x to y decimal places
sep='x'	Puts x between each concatenating item in an expression when printing
end='x'	Puts x at the end of each line when printing output of a for loop indexing an expression
input()	Collects input
range(x, y, z)	Works through from x to y (not inclusive of y), in steps of size z (if negative, works backwards)
len()	Length of anything
min()	Find minimum value
max()	Find maximum value
abs(x)	Absolute value of x
pow(x, y)	x to the power of y
variable.index()	Index of the first occurance of x in *variable* (at or after index i and before index j)
variable.replace(x, y/x, y, n)	Returns a copy of *variable* will all occurrences of x replaced with y (or just the first n)

Using External Packages

- First download the package (random, numPy, matplotlib, etc.)
- When using a package function in your code:

```
# To use the package using the nickname callerID
import package as callerID

# Using a function from the package in your code
callerID.function()
```

- You might want just one part of a larger package:

```
# Imports only that subpackage or specific function
from package import subpackage/function as callerID
```

Numeric Data

Numeric Operations

Command	Purpose
x + y	Sum of x and y
x - y	Difference of x and y
x * y	Product of x and y
x / y	Quotient of x and y
x // y	Floored quotient of x and y
x % y	Remainder(modulus) of x/y
-x	x negated
+x	x unchanged
x ** y	x to the power of y

Non-Numeric Data

Strings

Contructing a String

Command	Purpose
\n	Creates a blank line
\t	Create a tab
special character	Stop it from performing an action where you don't want it to (Escape the character)
"" \ ""	Used alone splits a string if you have too many character in a string to fit it on one line
("" "")	Don't need \ if strings are within brackets
""" """	Can use for very long strings, so you don't need to escape characters for quotes

- You can use variables in strings using an f-string

```
# Put the variable you want included in the string in {}
string = f'blah blah {variable}'
```

String Specific Functions

Command	Purpose
string.upper()	Returns a copy of *string* with all elements converted to upper case
string.lower()	Returns a copy of *string* with all elements converted to lowercase
string.strip(‘x’)	Return a copy of the string without x, or if x is unspecified, with whitespace removed (spaces, tabs, etc)
string.lstrip(‘x’)	Return a copy of a string with x removed if it's the first character
string.replace(‘x’, ‘y’, z)	Replace the first z number of elements that match x with y
string.isnumeric()	Return True if string has only numeric characters
string.isdigit()	Also returns True is string has only numeric characters
word[x]	Will look at element x in a string

- In strings spaces, and grammar count as elements, except \ when escaping special characters
- If using lower() and strip() together, it should be in order string.lower().split()
- To break a string which is a sentence, into a list of of the words as strings, ignoring any punctuation:

```
words = re.sub("[^\w]", " ", element)
```

- This can also have .lower() or whatever added to the end
- The character values are ranked from space, and other grammar, to letter in alphabetical order
- Concatenated string expressions wont have a any space between, so use + ’ ’ + to add a space

Conversion Between Variable Types

Command	Purpose
int(x, *optionalbase*)	Converts x to an integer. Base specifies the base if x is a string
float(x)	Converts x to a floating-point number
complex(*real*, *optionalimaginary*)	Creates a complex number
str(x)	Converts object x to a string representation
list(X)	Converts into a list of items
set(x)	Converts into a set of items
chr(x)	Converts an integer to a character
unichr(X)	Converts an integer to a Unicode character
ord(X)	Converts a single character to its integer value
hex(X)	Converts an integer to a hexidecimal string
oct(X)	Converts an integer to an octal string

Lists

- Can contain combination of variable types
- Can contain other lists
- Can be indexed

Command	Purpose
list = [x, y]	Creates a list containing x and y
list[n]/*list*[n][x]	Calls on element n in *list* / or element x of element n if n itself is a list
list[n] = x	Updates element n in *list* to be x
del *list*[n]	Deletes element n in *list*
list.append(x)	Adds x at the end of *list*
'x'.join(*list*)	Turns a list into a string separated by x
list.extend(x)	Adds a list to the end of a *list*
list.insert(x, y)	Inserts item x at position y
list.remove(x)	Removes first item from the list whose value is equal to x
list.count(x)	Counts how many times x appears in a list
print(*list*)	Prints everything in *list* combined, in order, separated by commas

Tuples

- Anything within brackets, separated by commas
- Immutable:
  - Cannot change length (no append or delete/remove)
  - Ordered
- Can be any type of information
- Created with:

```
# Standard way
variable = (a, b, c)

# Lazy way, auto stores in brackets
variable = a, b, c
```

- It is a sequence, so you can:
  - Use indexing
  - Use vector operations

Sets

- Defined with {}
- To create a set:

```
# Empty set
variable = set()

# Set of items
variable = {a, b, c}
```

- Not a sequence, so:
  - Unordered
  - No duplicates
    - {1, 2, 3} = {2, 3, 1} = {1, 2, 3, 3, 2}
- If you turn a list of string that are words into a set, it will separate each word into individual letters, to avoid this if you are pulling the words from something to a list, you can do it directly into a set instead with set() around everything else including .lower().split(), etc.
- if *set*: will be if there is anything in *set* == True, else: False
- Empty set {} = 0
- Can be written in predicate form:
  - {1, 2, 3, 4} = {x: x is a positive integer less than 5}
    - Where ':' represents | or "such that"
  - Or

Command	Purpose
set(*list*)	Turns a list into a set
set.add(x)	Adds x to an existing set
set.update(x)	Adds variable x holding multiple items to an existing set
set1.intersection(*set2*)	Find the intersection of *set1* and *set2*
set1.union(*set2*)	Find the union of *set1* and *set2*
any(*item* in *set1* for *item* in *set1*)	Like intersection, but gives a boolean
set1.difference(*set2*)	Finds the difference between *set1* and *set2*, directional (1.diff(2) = 1-2, and 2.diff(1) =

Dictionaries

- Keys map to values
- Dictionary is a set of key:value pairs
  - Can add, delete, overwrite
  - Actual keys themselves are immutable
- Created with:

```
dictionary = {a: b, c: d, e: f}
```

Command	Purpose
dict[*x*:*y*]	Adds the pair to the dictionary
del *dict*	Deletes pair from the dictionary
dict.keys()	Work with just the keys
dict.values()	Work with just the values
dict[*keyname*]	Will give the values for *key*
dict[*keyname*] += n	Will add n a numerical value held in *keyname*

- To get a dictionary with how many instances of each value there are, use Counter()

```
# This package comes with python
from collections import Counter

# set up word count
count = Counter()

# Run through a list, set, etc. and add them to the counter
count.update(variable)

# Or use with a statement
count.update(word for word in variable1 if word not in variable2)

# To print the x most common words in the count
count.most_common(x)

# To check if items are a key in the dictionary
if item in dict(count):
  do thing
```

Boolean and Comparison Operations

Command	Purpose
x in y	Returns True if x is in y
x not in y	Returns True if x is not in y
not x	If x is false, then True, else False
<	Strictly less than
<=	Less than or equal to
>	Strictly greater than
>=	Greater than or equal to
==	Equal
!=	Not equal
bool(*condition*)	Returns True if *condition* is met
isinstance(*variable*, *type*)	Returns True if *variable* is of type *type*
variable.startswith(x)	Returns True if *variable* starts with x
variable.endswith(x)	Returns True if *variable* ends with x
x or y	If x is false, then y, else x
x and y	If x is false, then x, else y

Control Structures

- Code within the control structures must be indented with ***4 spaces***
- One entry, one exit
- ***Do not use BREAK or CONTINUE***
- If you get stuck in an infinite loop, break with ctrl z or ctrl c
- *pass* - If you want to just continue with the next step
  - Can be used when testing a program you haven't finished, will avoid errors
  - Can be used to continue despite an error
- *continue* - If you want to continue a loop. This can be if you have a number of if elif statements for readability

if, elif, else

- Boolean expressions used to control the flow of a program
- Used -

```
if booleanexpression:
  do thing
elif booleanexpression:
  do thing
else:
  do thing
```

- List comprehension way
  - For use with a for loop

1. Just an if statement

```
newlist = [do thing to item for item in list
           if condition]
```

2. For if, elif and else

```
newlist = [do thing to item if condition
           else do thing to item if condition
           else do other thing to item
           for item in list]
```

While

- Repeat a block of statements until the condition is false
- Used -

```
while booleanexpression:
  do thing
```

- $variable$ += x in a while loop will add x to $variable$ each loop

For

- Create a loop that will repeat a set number of times
- Standard way -

```
for index in range():
  do thing to variable[index]
```

- Pythonic way -
    - Treats elements as items, so index will instead be the item at that index point, not the number used to identify the location itself
    - Can do traditional indexing using enumerate(), which will still use items, but will also track the index value and allows you to set the start value for the index while still working through the entire variable

1. Non-indexing

```
for item in variable:
  do thing to item
```

2. Indexing

```
for index, item in enumerate(variable, start = x):
  do thing to item
```

- List comprehension way -
    - Shortens the pythonic way to a one liner to create a list

```
newlist = [do thing to item for item in list]
```

- To do loops within loops

```
newlist = [[do thing to item for item in internal list] for internal list in list]
```

User Made Functions/Modules

- Set a function with -

```
def functionname(arguments):
  '''Purpose of function
  argument - description of argument'''
  do thing
  return output
```

- Indent everything within the function by 4 spaces
- There doesn't need to be any arguments, but you still need ()
- return(*variable*) will be the output of the function
- Without a path, to use, it will need to be in the same directory
    - To use -

```
# To use functions from a different script in the same file
from scriptname import *

# To use a function from the script
function()
```

- Use packages to collect modules together
  - Packages are a directory called ___*packagename*___.py (two underscores either side)
  - Would need to call with -

```
# To use function from package
from package import function

# Using function
package.function()
```

- If a function is made for use in a bigger program, you can make sure the main program only runs if it is called directly by making it a function called main() and put this if statement at the bottom of the program:

```
if __name__ == '__main__':
  main()
```

Object Oriented Design

- Create a file to carry out related tasks
- Then create classes to hold information for related objects and functions for use with those types of objects
- How to use:

```
class Noun():
  '''Purpose of class'''
  globalvariables = x

  def __init__(self, instanceinfo):
    '''instanceinfo - description of instance info'''
    self.instanceinfo = instanceinfo

  def verb(self, otherinfo):
    '''Purpose of function
    otherinfo - description of other info'''
    do thing to instance
    return output
```

- To use a class:

```
# To use the class
from program import Noun # OR import * for all classes

# Creating an instance of the class
object1 = Noun(instanceinfo)

# Using a class function on the class instance
object1.verb(otherinfo)
```

Class Relationships

- Classes can interact with each other (message passing)
- This can also be a part of a larger class
- Types of relationships:
  - Aggregation - Exist separately but one can be part of another
  - Composition - One class exists only when the one it is part of exists
  - Inheritance - One class is just the first class, only with some adjustments
  - Other
- How to use:
  - Aggregation:
    - The first class is created completely independently
    - The second class is created independently, but calls on the first class in a module to use the functionality of that class
      - Usually it is the class of which the other is a part that is doing the calling
- Composition:
  - Created exactly the same as aggregation, however the class that is part of the larger one is useless/cannot exist, without being used as part of the other class
- Inheritance:
  - The first class (parent/superclass) is created independently
  - The second (child/subclass) class is created as:

```
# Parent class needs to exist, so it must be made first
class parent():
  # Parent class stuff

class child(parent):
  '''Describe child function'''
  childglobalvariables = x

  # If using init information from the parent plus some arguments in the child
  def __init__(parentarguments, childarguments):
    '''Explain'''
    super.__init__(parentarguments)
    # Don't need to write self.parentarguments = parentarguments
    # - it's pulled from the parent function
    self.childarguments = childarguments
```

- The child will replicate and use the parents functions if amending/adding to them with:
  - def *same name as parent function*(*same arguments*, *plus new arguments*)
  - super().*parent function*
    - Then make any changes
- If using the parents functions as is, the child does not need to call on them within the class at all, they can just be used by the child

Exceptions

- When something goes wrong, it said that an exception is "thrown"
  - If it has code to stop the error from causing issues, it is a "caught" exception
  - If it is not caught, then when it causes issues, it is called an error
- You can specify a number of specific errors, and have a catch all generic exception, but might not want a catch all
- In use:

```python
try:
  do thing
except exceptiontype:
  do thing
except otherexceptiontype:
  do thing
else:
  do thing
```

- For input checking functions:

```python
n = 0

while n == 0:
  try:
    variable1 = input(input)
    variable2 = int(variable1) # or float, string, whatever
  except exceptiontype as e:
    do thing
  else:
    do thing
```

- Raising exceptions allows you to specify in a class or function what to print if there is a certain type of exception, and then in the program or function calling on it, it will be able to print that:

```python
# What to write in the class
if boolean:
  raise exceptiontype("what to print")

# In the program using the class
try:
  thing
except exceptiontype as e:
  print(e)
  do thing
```

Exception Types

Exception	Meaning
Exception	Catch all
TypeError	Variable is wrong type for use in operation
ValueError	Variable is right type, but wrong value
IndexError	Element outside index range being called on
FileNotFoundError	Tried to use non-existant file
NotImplementedError	Class's method needs to be implemented in a child class

Files

- Need to create a file object to utilise the file contents
  - Use -

```
filevariable = open('path filename.filetype', 'x')
```

- Where 'x' is how to process the file/file modes

File modes

Read	Write	Append	Description
r	w	a	Read/write/append text files
rb	wb	ab	Read/write/append binary files
r+	w+	a+	Opens for reading and writing
rb+	wb+	ab+	Opens for reading and writing

- Need to close files safely
  - Use -

```
filevariable.close()
```

- Defaults:
  - Without a specified path, files will only be searched for in the current directory
  - Default for opening files is 'r'
- The pythonic way -

```
with open('filename', 'x') as f:
  filevariable = f.y() # See reading files for options for y
```

- With exception handling:
    - This is best practice for handling files -

```
try:
  with open('filename', 'x') as f:
    filevariable = f.y()
except OSError as e:
  print('Error with file open(): ', e)
except:
  print('Unexpected error: ', e)
```

Reading Files

- Three types:

Type	Purpose
filevariable.read()	Reads entire contents of file as a single string
filevariable.readline()	Reads only up until first instance of \n
filevariable.readlines()	Reads entire contents of file and creates a list of strings split where there is an \n

- You can turn text files into a list of the words it contains with -

```
punctuation = '~!@#\$\%^&\*()\_+{}|:"<>?`=[]\\;\',./'
book = filevariable.translate(str.maketrans(",", punctuation))
words = book.lower().split()
```

Writing/Appending Files

- .write() will overwrite anything in the file, with whatever is within the brackets
- .append() will add whatever is within the brackets to the end of what is already written in the file
- Done when opening, and still needs to be closed if not using pythonic method

CSV Files

- A type of text file
- Lines separated with \n
- Line elements separated with , (commas)
- Can read and write with pandas -

```
# To read
dataframe = pd.read_csv("filename.csv", header = 0, index = False)

# To write
#   header = 'columnnames' to transfer the keys
#   index = True or False depending if you want an index in the file
dataframe.to_csv('filename.csv', header = 'columnnames', index=False)
```

- It automatically turns it into a dataframe, no splitting, etc. needed

System Calls

- Work with directories within a program
  - Use -

```
import os
```

- Functions:
  - mkdir(*string*)
  - listdir()
  - chdir(*string*)
  - getcwd()
  - rename(*source*, *destination*)

Command Line Arguments

- Use -

```
import sys
```

- sys.argv is a list of all the command line arguments
- Use sys.argv[*n*] to use the nth argument entered

Parameter Sweeps

- Done in shell scripts
- Commands
  - Receive command line input
  - Can also give command line input
    - Uses this to run through another program numerous times

```
python filename $n > outputfile
```

- Loops through all permutations of values

Regular Expressions

- regex101.com - good for these
- Use -

```
import re
```

- Cleans up inconsistent files read in
- Flexible matching
- Metacharacters

Command	Purpose
[ ]	Set of characters to match ([cbm]at = cat, bat, mat)
ˆ	Gives complement ([ˆ5] = not 5)
\	Special sequences, or escape
\d	Matches to any decimal digit [0-9]
\D	Matches to any non-decimal digit
\s	Matches any whitespace character [\t\n\r\f\v]
\S	Matches any non-white space character
\w	Matches any alphanumerica character [a-z, A-Z, 0-9]
\W	Matches any non-alphanumerica character
.	Matches anything other than newline
?	Matches to 0 or 1 repeats of the last
	Matches to 0 or more repeats of the last (ca*t = ct, cat, caat, etc)
+	Matches to 1 or more repeats of the last (ca+t = cat, caat, caat, etc)
{*m*, *n*}	Matches at least *m* repeats and at most *n* repeats of the last

- **Note** - Don't create groups for spaces, don't need brackets
- Can combine metacharacters
  - .at - Matches anything with a character followed by "at"
  - [0-9][a-z] - Matches any digit followed by a lowercase character
  - [0-9]\s[a-z] - Matches any digit followed by a whitespace, followed by a lower case character
- Use r'*patten*' as the argument in all methods
- Use re.VERBOSE in the arguments with r' ' '(*pattern1*) \n (*pattern2*) to improve readability
  - Can use comments inside due to being verbose
- Search commands:

Command	Purpose
compile	Create pattern object for reuse as *compv.function*()
match()	Match to beginning of string
search()	Returns a list of matches
findall() returns a list of matches	
finditer()	Finds an iterator of matches

- Search commands all create objects

- Can use further functions on the objects:

Command	Purpose
group()	Gives you everything you matched with
group(*n*)	Gives you a block of the match aligning with that chuck in the expression
start()	The starting position of the match
end()	The ending position of the match
span()	A tuple containing start and end

Generating Random Numbers

- Use -

```
# To use random
import random

# To use random function
random.function()
```

Command	Purpose
random()	Returns the next random floating point value from the generated sequence ($0 \leq \text{n} < 1.0$)
seed(*n*)	Use before generating numbers, to ensure the same values will come up everytime you run the code (repeatable)
randint(*x*, *y*)	Gives you a random integer between *x* and *y* (inclusive)
randrange(*x*, *y*, *z*)	Allows for steps (not inclusive of stop value)
choice()	Makes a random selection from a specified sequence

Numpy Arrays

- Use -

```
# To use numpy
import numpy as np

# To use numpy function
np.function()
```

- To create an array:

1. Directly -

```
np.array([x, y, z])
```

2. From a list - np.array(*list*)

```
np.array(list)
```

- Can use dtype=*type* to as an argument to specify what type of data you want the array to store the data as

3. You can make preset arrays of values:

- np.zeros(x) - Array of x 0's
- np.ones(x) - Array of x 1's
- np.fill(x, y) - Array of size x, with each element containing y
- np.random.random(x) - Array of x random numbers
- np.arange(x, y, z) - Array created using values within a range, specified the same as range()
- np.linspace(x, y, z) - Array of size z, of values between x and y inclusive, with each value evenly spread across the range
  - Be conscious of ranges for this one as it is inclusive
- You can loop and slice arrays the same as any other vector

Multidimensional Arrays

- To create a multidimensional array:

1. Build it from lists

```
variable = np.array([[x, y], [a, b]])
```

- So you set up an array with a list of lists separated by commas
- Each list is one row, and the comma indicates the beginning of the next row

2. Build using array builders

```
variable = np.zeros((x, y))
```

- This creates an array with x rows, and y columns
- Can use any array builder (ones, etc.)

3. Use meshgrid, which is similar to arange for 1-D arrays

```
x, y = np.mgrid[0:a, 0:b]
```

- This creates two arrays
- x will have a rows, with row 0 containing b 0s, row 1 containing b 1s, etc.
- y will have 5 columns, with column 0 containing 5 0s, column 1 containing 5 1s, etc.
- All versions create a list within a list and will be shown as:
  - [[x, y], [a, b]]
- Indexing is similar, $array$[x, y] will call on the element where row x meets column y

- Slicing is the same method, you just put in a specification per dimension separated by commas and the output will be the intersection of the specifications
  - eg. $array$[a:, ::b], will give you where rows a to end, intersect with every b columns

Looping Through Multidimensional Arrays

- Need a loop for each dimension, with that loop only specifying one dimension at a time
- This can be done with -

```
for i in array[:, 0]:
  for j in array[0, :]:
    do thing
```

- This would run through every row (:), and ignore columns
- Then would ignore rows, and run through every column (:)
- The pythonic way:

1. Non-Indexing -

```
for row in array:
  for item in array:
    do thing to item
```

2. Indexing -

```
for rindex, row in enumerate(array):
  for cindex, item in enumerate(row):
    do thing to item
```

- In this case, rindex, and cindex represents the value of the index for the rows and columns respectively
- Enumerate is used to index the array in this way

Numpy Array Operations and Functions

- Describing Arrays

Function	Use
print(x)	Will print as a list of lists
np.size(x)	Tells you how many elements in the array total
np.shape(x)	Tells you (a, b), where a is number of rows, and b is number of columns
len(x)	Tells you the length of the first dimension (number of rows)
array.reshape(x, y)	Will break up *array* into x rows, and y columns

- Element-wise operations:

Command	Purpose
a + *b*	Adds elements of *a* to the associated element of *b*
a + *n*	Adds *n* to each element of *a*
a - *b*	Minus elements of *a* from the associated element of *b*
a * *b*	Multiply elements of *a* with associated element of *b*
a / *b*	Divide elements of *a* with associated element of *b*

- Or you can compare the elements of one array with those of another using a boolean comparison, which will return an array of True and False
- Rows and columns must be the same length
- Not using matrix rules
- Element-wise functions:

Command	Purpose
np.sqrt()	Square root of each element
np.sin(), cos(), etc	Trig operation on each element
np.exp(), log(), etc	Mathematic operations on each element
np.add(), minus(), multiply(), divide(), etc.	Standard maths on each element

- Array-wise functions:

Command	Purpose
variable.sum()	Sum of array elements
variable.min()	Minimum value in the array
variable.max()	Maximum value in the array
variable.mean()	Mean of the array elements

- For multidimensional arrays, array-wise functions can be used across the entire array, or you can slice it to be used for only one dimension of the array
  - eg. *array.function*() - Will combine the elements from the entire array
  - eg. *array*[:, 0].*function*() - Will only combine the elements from the rows in column 0
- **Need to be careful, sometimes you might get an inexact value due to the translation of binary to decimal**

Matrices

- Again need to use numpy
- Set up using matrix() function

  1. Using lists -

```
variable = matrix([[x, y], [a, b]])
```

2. Using formatting like matlab -

```
variable = matrix('x y; a b')
```

- Can then use matrix manipulations
  - Operations (+, *, **) will be matrix appropriate
  - Can use np.linalg (linear algebra package)
    - np.linalg.det(*array*) - Gives the determinant
    - *array*.T - Transposes a matrix
    - x, y = np.linalg.eig(*array*) - Will assign x the eigen value and y the eigen vector

Grids

- Also use numpy, and often matplotlib (see below)
- Group of cells that affect those around them
  - Those affecting the current element/cell or "site" are it's neighbourhood
  - Types of neighbourhood:

1. VonNeumann

NW	**N**	NE
W	***Site***	**E**
SW	**S**	SE

2. Moore

NW	**N**	**NE**
W	***Site***	**E**
SW	**S**	**SE**

- The element/cell in use, or "site" is calculated in relation to the neighbourhood with:

(ROW - 1, COL - 1)*formula*	(ROW - 1, COL)*formula*	(ROW - 1, COL + 1)*formula*
(ROW, COL - 1)*formula*	(ROW, COL)*formula* (***Site***)	(ROW, COL + 1)*formula*
(ROW + 1, COL - 1)*formula*	(ROW + 1, COL)*formula*	(ROW + 1, COL + 1)*formula*

- Grid algorithm (needs matplotlib.pyplot, see below):

```
# Function for use in algorithm
def function(n, m):
  '''Define'''
  do thing # for all cells part of the neighbourhood
  return output

# Initial values
rows = n
columns = m

initialgrid = np.zeros(n, m)
affectingpoint = value

initialgrid[a, b] = affectingpoint
resultinggrid = np.zeros(n, m)

# Running algorithm
for timestep in range(x): # where x is the number of time periods
  for n in range(0, n):
    for m in range(0, m):
      resultinggrid[n, m] = initialgrid.function(n, m) # Run that value through
                                                       # the function outlined
  resultinggrid[a, b] = affectingpoint # If you need the affecting point at
                                       # the original value
  initialgrid = resultinggrid
plt.imshow(resultinggrid, other)
plt.show
```

Matplotlib

- Use -

```
matplotlib.pyplot as plt
```

- Plot types:
  - plot(x, y) - Plot x on the x axis, and y on the y axis (default for single is y axis)
  - bar(x, y) - Plot a bar graph
    - width = n argument from 0 to 1 if you want some spacing between bars
  - hist(x) - Plot a histogram
    - Default breaks data into 10 bars, use bins=n to change to n bars
    - Cumulative = TRUE for cumulative histogram
    - histtype='step' to use a line instead of bars
    - normed=True to normalise the data

- Plotting tools:

Command	Purpose
title('*Title*')	Main title for graph
xlabel('*Label*')	Label for x axis
xaxis(x, y)	Sets start and end of x axis
ylabel('*Label*')	Label for y axis
yaxis(x, y)	Sets start and end of y axis
show()	The final command to print the graph with a collation of prior commands
plot.get_figure()	Collect figure information
plot.savefig(*filename.filetype*)	To save plot image
plt.savefig(*filename.filetype*)	To save plot before/instead of plt.show()
filenames	Can use variable names, + to concatenates with strings or other

- Multiples:
  - Just plot all the plots before using show()
  - Make multiple graphs side by side using subplot() -

```
plt.figure(1) # Makes it one figure
plt.subplot(x, y, z) # Where you want to place the graph in terms of a matrix
```

- x is how many rows of subplots you want
- y is how many columns of subplots
- z is the position you want this subplot, counting from 1 and 1,1 of the matrix of subplots, and counting from left to right, returning to the left as you move down a column
- Graph visuals:
  - You can use styles (affects all future plots if not used in a with statement) -

```
with plt.style.context(('style')):
  plt.plot() # do usual plot stuff
```

- '*a b*' argument to change to *a* coloured *b* shaped dots
  - Colour shorthand examples:
    * b - blue, g - green, r - red, etc
  - Marker shape examples:
    * +, „ ., 1, 2, s - square, ^ - triangle, etc
  - Linestyles examples:
    * -, –, -., :, 'steps', . . . , etc
- Or the argument color='*colour*'
  - 'blue', 'pink', etc
- grid = TRUE - argument to have a grid
- alpha=n - argument between 0 and 1 to set opacity if you want to overlay data
- interpolation = 'bilinear' - argument to smooth the resulting graph, data is no longer true data

- **Notes:**
  - You can save any generated image with -

```
figurename = plot.savefig('filename.filetype')
```

- Can use an f-string to include variables in the file name
- **mpl_toolkits.basemap no longer exists! It is now cartopy**

Plotting Multi-Dimensional Data

- Also with matplotlib
- Warning for colour maps: With spectrums, some colours are brighter than others, which can make the plot misleading
- Contour plots

```
plt.contourf(x, y, f(x, y), n, aplha=a, cmap=plt.cn.b)
```

- Contour function is a 3 dimensional plot which is filled with colour based on b
- x is x-axis value
- y is y-axis value
- f(x, y) is the function for the third dimension
- With $n + 1$ levels/layers/depths of colour contrast
- And a transparency (0.0 - 1.0)

```
plt.contour(x, y, f(x, y), n, colors='a', linewidth=b)
```

- Contour is a 3 dimensional plot which has lines whose colour is based on a
- The lines with have a width of b
- Scatter plots

```
plt.scatter(x, y, f(x, y), s=a, c=b, alpha=c)
```

- Where a is the type of point to use
- And b is the colour-scheme to use

Scipy N-Dimensional Images

- Use -

```
from scipy import ndimage
```

- Plot the images using a similar function to matplotlib -

```
plt.imshow(imagename)
```

- Use a colourmap with -

```
plt.imshow(imagename, cmap=plt.cm.colourscale)
```

- Plots the same as a normal plot, except that 0 for the y axis starts in the top left, rather than the bottom left
- Functions:
  - ndimage.shift($imagename$, $(x.\ y)$) - Will have the top left corner at coordinates (x, y)
  - ndimage.rotate($imagename$, x) - Will rotate the image n degrees anti-clockwise
  - $imagename[slice, slice]$ - will crop the photo by keeping only what is within the specified slices for each dimention
  - To pixelate the image, just slice it with steps. The more steps, the more pixelated

Pandas

- Use -

```
import pandas as pd
```

- Used to create and manipulate dataframes

Pandas Dataframes

- Uses standard indexing (0-based)
- Essentially a dictionary:
  - Keys are column labels
  - Value is a row of values associated with each column
- Creating a dataframe -

```
df = pd.DataFrame(['Label A': [lista], 'Label B': [listb], 'Label C': [listc]])
```

$\rightarrow$

Index	*'Labela'*	*'Labelb'*	*'Labelc'*
0	a1	b4	c7
1	a2	b5	c8
2	a3	b6	c9

- Renaming columns:

```
dataframe.rename(columns={
  'current_name': 'new_name',
  'Current_name2': 'new_name2'
}, error='raise')
```

- Dataframe functions:

Command	Purpose
df.copy()	Creates separate object, so changes wont affect original
type(*df*)	Will tell you it's a class defined as pandas dataframe
df.types	Will give you the type of the values in each column
set/list(*df*['*key*'])	Creates a set or list from column entries
df.columns	Gives a list of key names (column names)
df.columns.values	Turns each column into an array assigned to its key
df[*key*]	Allows you to access the array assigned to *key*
df.describe()	Gives count, mean, std, min, 25%, 50%, 75% and max
df.shape	Gives the total number of keys and values
df.head(n)	Gives the first n rows, or first 5 if n undefined
df.tail(n)	Same as head, but for the last rows
pd.unique(*df*[*key*])	Creates arrays with the unique values in that column as the key for each
df.min()	Min
df.max()	Max
df.mean()	Mean
df.std()	Standard deviation
df.count()[x]	How many elements, or how many x's
df.groupby(*key*)	Groups the dataframe based on unique instances in *key*
df.groupby(*key*)[*alluniquekey*].count()[x]	Counts all unique instances of *alluniquekey* that match x in grouped data
df[*key*] array operation	Perform array-wise maths
df.plot(kind = '*plottype*')	

Indexing/Subsets Pandas Dataframes

- Slicing is done the same, but accounts for using labels
- loc function is used when idexing rows, but calling columns by name
- iloc function used when indexing both rows and columns
- New dataframes can be created from a subset of another dataframe
  - This can be done with a boolean based on values in a column

```
# Create dataframe of only the rows where the column Booleans has True
#   as the value in that row
new_df = old_df[(old_df['booleans'] == True)]
```

- Or based on labels using drop()

```
# Removes columns with keys in the list
dataframe.drop(columns=[key1, key2])

# Or removes a column or row based on a label
#   axis =
#     0 = rows
#     1 = columns
#   inplace=True means everything remaining stays in their current order
dataframe.drop('label', axis = 1, inplace=True)
```

- Indexing/subsetting:
  - Rows for all columns
    - Range of rows - $df\,[rowindexing]$
    - Specific rows - $df.iloc[[rowlist], :]$
  - Columns for all rows
    - Range of columns - $df\,.iloc[:,\ columnindexing]$
    - Specific columns - $df\,.loc[:, [keylist]]$
    - Specific columns by label - $df\,[[keyslist]]$
  - Rows and columns
    - Range of rows and columns - $df\ iloc[rowindexing,\ columnindexing]$
    - Specific rows and columns - $df\ loc[[rowlist], [keylist]]$
- Can use booleans to create subsets
  - $df\,[df.key$ boolean $value\ in\ column]$

Using Functions on Dataframe Elements

- To apply a function to every element of the dataframe:

```
# Function to be used
#   If you use return, you can either change the elements in the dataframe
#     or if you assign a variable to the dataframe as you run the function,
#     a new dataframe will hold the results
#   If you don't use return, nothing happens to the dataframe, but you can
#     add to a list or something like that based on each element

def df_function(element):

  result = do thing to element

  return result

# Run function on all elements
#   lambda x: is just saying that each individual element is assigned as
#     x for use in the function argument
#   na_action='ignore', means that any elements will be ignored instead of
#     returning an error
new_dataframe = dataframe.applymap(lambda x: df_function(x), na_action='ignore')
```

- To apply a function to every row or column of the dataframe:

```
# Treats row or column as an individual series and will work on the entire
#   series at once, not element by element
# Function to apply
def row_or_column_function(row_or_column):

  # If you want to treat row as series
  do thing to row_or_column

  # If you want to access elements of the row
  for element in row_or_column:
    do thing based on element

# Run function on dataframe
# lambda x: is saying each row or column is assigned as x for use in the
#   function argument
# axis = assigned whether it runs on columns or functions
#   0 = columns (default)
#   1 = rows
dataframe.apply(lambda x: row_or_column_function(x), axis = )
```

Plotting in Pandas

- Based on matplotlib.pyplot

```
plot = data.plot(kind = 'type', title = ('title'), legend = None)
  plot.set_xlabel('xlabel')
  plot.set_ylabel('ylabel')
```

- Use same method as matplotlib to use styles

Seaborn

- Use -

```
import seaborn as sns
```

- A package for data visualisation
- Also based on matplotlib -

```
sns.set_style("style") # if using a style
plot = sns.barplot(x = xvalues, y = yvalues, palette = 'palette')
plt.xsticks(rotation = 90) # to turn x labels on their side
plt.show()
```

Bokeh

- Use -

```
from bokeh.plotting import figure, output_file, show
from bokeh.palettes import colour/s # colours as lowercase for functions, uppercase
                                    # for dictionary and end in number based on size
                                    # of set being used
from bokeh.transform import factor_cmap # for colour maps
```

- Creates extra fancy plots apparently, outputs them as html
- **Note: bokeh.charts no longer exists**
- Need to set it to output the plot as a html image first -

```
output_file("filename.html")
plot = figure(title = 'title', x_range = xdata # if x's are categorical, not needed for
                                               # numerical x's)
plot.xaxis.axis_label = 'xlabel'
plot.xaxis.major_label_orientation = radians # To change angle of the x axis labels
plot.yaxis.axis_label - 'ylabel'
plot.vbar(x, top = y, width = width, color = colour)
show(plot)
```

- **Other lecture examples not valid - did not look up look up how to correct**
- Can use with holoview package (it does still exist!), but lecture examples not valid - did not look up how to correct

Plotly

- Just use instead, easy to work with, best documentation (java based)
- To use for an animated timescale -

```
import plotly.express as px

# Create graph
fig = px.scatter(graph,
                 x="Row",
                 y="Column",
                 animation_frame="Year",
                 animation_group="Name",
                 color="Species",
                 hover_name="Name") # Using preset variables for
                                    # each and auto grouping where needed
fig.show()
```

Spyder

- Instead of vim, interactive, like RStudio
  - Except it's shit
- To use

```
$ spyder&
```

- Press F8 to run through the code

Working with MySQLdb Databases

- Most databases written in SQL
- Implementation of MySQLdb:
  - Initiate -

```
import MySQLdb
db = MySQLdb.connect('localhost', 'user', 'file', 'database')
```

- Prepare cursor object

```
cursor = db.cursor()
```

- Execute SQL query

```
cursor.execute("SELECT VERSION()")
```

- Fetch a single row

```
data = cursor.fetchone()
```

- Disconnect from server

```
db.close()
```

- Drop table if it already exists

```
cursor.execute("DROP TABLE IF EXISTS X")
```

- Create table

```
table = """CREATE TABLE (INSTRUCTIONS)"""
cursor.execute(table)
```

- Insert a table

```
table = "INSERT TABLE (COLUMNS) \ VALUES ('%ROWS') % \ ('ROWVALUES')"
try:
  cursor.execute(table)
  db.commit()
except:
  db.rollback()
```

- Extract data

```
items = cursor.fetchall()
for row in items:
  a = row[n]
```

Web Scraping

- HTML (web pages) have various tages described below:
  - <!DOCTYPE html> - Start with a type declaration
  - Document contained between
  - Visible part of the document is between
  - Heading are defined with tags
  - Paragraphs are defined with the
    tag
  - Links are defined with
  - Tables are defined with
    , row as
    and rows are divided into data as
  - Lists start with
    (unordered) and
    (ordered), each item of the list starts with

- Accessing and parsing a web page -

```
import urllib.request
```

- Using BeautifulSoup as API -

```
from bs4 import BeautifulSoup
```

- Specify the URL -

```
site = "link"
```

- Collect html from the URL -

```
webpage = urllib.request.urlopen(site)
code = BeautifulSoup(webpage, 'html.parser')
```

- Make code readable -

```
print(soup.prettify())
```

- Extract a table

```
table = code.find('table', class_='table type')
rows = table.find_all("tr")
for row in rows:
cells = row.find_all('td')
a.append(cells[n].get_text())
```

Data Cleaning

- Basic data cleaning:

```
import pandas as pd
import numpy as np

def clean(data):
  data = data.replace(item, np.nan) # if need to make some values nan
  data = data.dropna (axis = 0, how = 'any') # 0 is for rows, 1 for columns
  data['*Date Time*'] = pd.to_datetime(data['*Date Time*'])
  return data
```

- Combining data sources

```
sensor = rd.read_csv('filename')
sensor = clean(data = sensor)
other = pd.read_csv(otherfile)
other = clean(data = otherfile)
start = 'datetime string'
end = 'datetime string'
sensor = timeframe(start, end, sensor)
other = timeframe(start, end, sensor)
ax = sensor.plot(x, y)
other.plot(x, y) # plots them together
```

Cloud-Based Dashboard Services

- With InitialState
  - Code to push to InitialState

```
from ISStreamer.Streamer import Streamer
ACCESS_KEY = 'key'
BUCKET_KEY = 'key'
BUCKET_NAME = 'name'
streamer = Streamer(bucket_name=BUCKET_NAME, bucket_key=BUCKET_KEY,
                    access_key=ACCESS_KEY) # Creater streamer instance
```

- Send data

```
streamer.log("label", info)
```

- Close the stream

```
streamer.flush()
```

Object Detection

- Example -

```
import argparse
import cv2
```

- Parse the arguments

```
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required=TRUE, help="path to the input image")
ap.add_argument("-c", "--cascade", default="haarcascade_frontalcatface.xml",
                help="path to cat detector harr cascade")
args = vars(ap.parse_args())
```

- Load input image (in greyscale)

```
image = cv2.imread(args["image"])
```

- Load detector Haar cascade, then detect input image

```
detector = cv2.CascadeClassifier(args["cascade"])
rects = detector.detectMultiScale(colourscale, scaleFactor=n, minNeighbours=m, minSize=(a, b))
```

- Loop over images and draw rectangles

```
for (i, (x, y, w, h)) in enumerate(rects):
  cv2.rectangle(image, (x, y), (x + w, y + h), (0, 0, 255), 2)
  cv2.putText(image, "Cat #{}".formate(i + 1), (x, y - 10),
              cv2.FONT_HERSHEY_SIMPLEX, 0.55, (0, 0, 255), 2)
```

- Show detected faces

```
cv2.imshow(image)
cv2.waitKey(0)
```

- Will print the image with rectangles over the cat faces

More Complex Functions

- Format to 2 decimal places

```
\${:.2f}\\n".format(numericvariable)
```

- Delay output by n seconds

```
import time

time.sleep(n)
```