~python~ "learn to fail, or fail to learn."

GITHUB

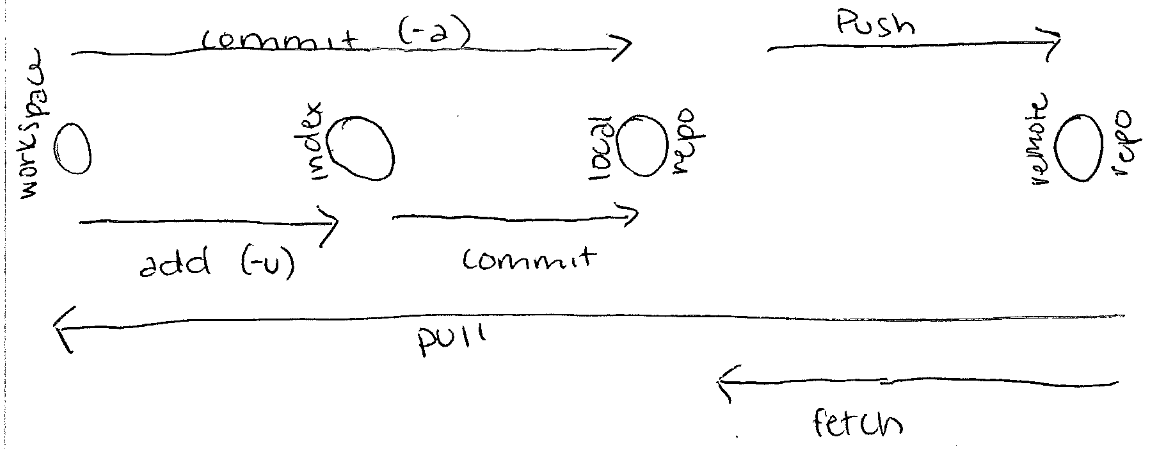
[Maintaining files:

- stored in local repo directory, "workspace"
  - to commit a change:
    - "add" = add file to the index (staging area indic
      files from workspace that should be updat
    - "commit" = after files are in index, commit index -
      local repo
- changes made to local repo do not affect remote repo & vice versa

- "push" = send local repo to remote repo

- "pull" = retrieve remote repo to local & workspace

- Flow chart:



]

[Git commands:

- create new remote repo = "git init <PATH> --bare"
- copy remote repo to local = "git clone <URL> <PATH>"

- "git status" = current status of local repo
- "git log" = log of all commits to repository
- "git add <FILE>" = add file from workspace to index
- "git commit" = add all files in index to local repo
- "git push" = push local repo to remote
- "git pull" = pull remote repo to local]

Previous file versions:

- "git show <HASH>:<FILE>" = display contents of specific version of file. HASH is shown in log.
- "git checkout <HASH> <FILE>" = can change parts of file workspace
- "git diff" = will show differences between workspace and local

FUNCTIONS

[terminology :

- definition - actual code of function.
- call - to run the function by referring to it in code.
- return - send single value back to line calling function. For, multiple values return a collection.]

[Syntax :

```
def < function name> (<parameter 1>,<2>,<3>. . . ,<n>) :
#1 indent <line 1>
          <line 2>
          <line n>

          return <value>                    ]
```

[recursion (example):

```
def mystery (num):
#this line is base case
          if num == 0:
               return 1

          return num * mystery (num-1)
                                                  ]
print (mystery (4))
```

[Parameters :

- arguments should be supplied in same order as params listed in definition, unless params names used when calling function.
- arguments must be provided, unless param has default.]

[Scope :

- local variables - those created within a function
- global variables - those declared outside a function
- local variables cannot be accessed outside function]

COLLECTIONS

- " = " = points variable name into a memory location
- tuples:
  - order maintained
  - objects can be any type
  - immutable
  - "(<item 1>, <2>, <3>. . . <n>)" = to create
  - "tuple(<item 1>, <2>, <3>, . . . <n>)" = to create

- list:
  - order maintained
  - objects can be any type
  - mutable
  - "[<item1>, <2>, <3>, ..., <n>]" = to create
  - "list(<item1>, <2>, <3>, ..., <n>)" = to create
  - methods:
    - "<list>.append(<item>)" = add item to end of list
    - "<list>.~~insert~~ remove(<item>)" = remove item from list
    - "<list>.insert(<position>, <item>)" = insert item to list
    - "<list>.extend(<list2>)" = add all items from list 2 - list
    - "<list>.pop()" = remove last item from list
    - "<list>.reverse()" = reverse list ordering

- operations and methods:
  - "<list>[<index#>]" = selects item # in tuple / list
  - "<list>[<start#>: <stop#>]" = creates sublists / subtuples

- "<item> in <list>" = to check for item in list/tuple
- " len(<list>)" = will return # of items in list/tuple
- "<list>.count(<item>)" = will return # of times item is found
- "sorted(<list>,[reverse=True])" = function will sort and return as a list]

[aggregation functions:

- both tuples & lists support:
  - " sum(<list>) " = returns sum of all items in list (numeric type)
  - " min(<list>)" = returns min value of items in list
  - " max(<list>)" = returns max value of items in list]

[Splitting and joining:

- splitting = create a new list of strings from single string separated by a delimiter.
  = if no delimiter, default spits at any white space.
  = can have multiple delimiters (characters).
  = "<string>.split([delimiter])

- joining = combine a list of strings back into a single string.
  = delimiter must be specified.
  = delimiter can be multiple characters.
  = "<delimiter>.join(<list>)"]

[Iteration:

- Both tuples and lists can be iterated across using "for" statement.
- example:

```
coordinate = (5, 2, 1000)

for axis in coordinate:
```

```
# 1 indent next lime
    if axis > 100 :
#2 indent  next line
        print(" warning , {0} is  greater  than  100!" . format(axis))
# 1 indent nex line
    else :
# 2 indent next line
        print("OK: {0}" . format (axis))
```

sets :

- collection of items
- unordered
- mutable, items can be add & removed
- cannot add existing items (only unique)
- Syntax = " set (<item 1> , <item 2> . . . <n>)"
- operations and methods :
  - "<set> .add (<item>)" = add item to set
  - " <set> . remove (<item>)" = remove item from set
  - " <set> . discard (<item>)" = remove item if present (no error if not found)
  - " len (<set>)" = # of items in set
  - " <item> in <set> " = checks for existence of item in set
  - " <set1> . issubset (<set2>)" = check if every item of set 1 is in set 2
  - " <set 1> . is superset (<set2>)" ↓ = check if every item of set 2 is in set 1
  - "<set 1> . union (<set 2>)" = new set with items common between and uncommon (all) in set 1 and 2
  - "<set 1> . intersection (<set 2>)" = new set with items only common between sets 1 and 2

- "<set1>.difference(<set2>)" = new set with items in Set1 that are not in set 2
- "<set1>.symmetric_difference(<set2>)" = new set with items unique to either set 1 or 2

Iteration:

- using "for" statement.
- order of iteration is unpredictable.
- set can converted to a list for predictable iteration, using sorted function.

dictionaries:

- collection of items
- unordered
- all items have associated "key" and are known as values
- key:
  - can be many data types (including tuples)
  - data types of keys and values dont need to match
  - must be unique in a dictionary
  - can only store 1 value each, but value can be a collection
- initialization:
  - "{}"
  - "dict()"
- populating:
  - "{<key1>: <value1>, <key2>: <value2>, ..., <key n>: <value n>}"

- operations and methods:
  - "<dict>[<key>]" = values retreived associated with key
  - "<dict>[<key>] = <value>" = set values to key
  - "<dict>.pop(<key>)" = remove key and associated value and returns value the key had.

- "len(<dict>)" = number of items (key/value pair) in dictionary
- "<key> in <set>" = check for existence of key in dictionary
- "<dict>.get(<key>, <default>)" = returns associated value if key exists, otherwise returns default value
- "<dict>.setdefault(<key>, <default>)" = if key is in dictionary, ret its value. If not, create key with default value.
- "<dict>.keys()" = return read only view of all keys
- "<dict>.values()" = return view of all values
- "<dict>.items()" = return view of key/value pairs

- iteration:
  - iteration returns key/value pair associated key which can be used to retrieve value
  - ".keys()", ".values()", and ".items()" can be iterated against

CLASSES

overview:

- classes are templates / definitions for instance objects
- instance objects can contain both:
  - instance (variables) / attributes
  - instance methods (functions)
- instance attributes and methods defined in the class relate to purpose of class

instantiation:

- instantiation of a class takes "template" for an object and creates object (stored as instance of class).
- classes themselves don't store information, instance object does.

purpose:

- keeps related functionality bundled together
- encapsulation - allows an object to be given as an argument or returned from a method.
- operator overloading - allow the print function to be used directly on objects

instance attributes:

- variables that are stored per instance object
- name of the attribute will be the same across all objects of the same class
- when an instance refers to its own attributes, use "self"
  - "self.<attribute>"
- when defining an instance method, first parameter mustbe "self"

syntax:

```
#define the class
Class <Classname>:
#init method required if class has attributes
        def_init_ (self, <parameter 1>,...,<parameter n>):
                self.<attribute name1> = <initial value>
                self.<attribute name2> = <initial value>

        def <method name> (self, <parameter 1>,...,<parameter n>):
                #body of method

#create an object instance of the class
<instance name> = <class name>()
```

object usage:

- after creating an instance object, the attributes and methods can be accessed using dot "."
  - "<object name>, <attribute>"
  - "<object name>. <method>()"
- default, an object's attribute can be set from both within and outside of the class

class attributes and methods:

- class methods do have "cls" as first parameter
- class methods have "@classmethod" decorator

operator examples:

- "__str__" = executed when instance object is cast into a string
- "__add__" = executed when your object has something added to it

- "__eq__" = executed when your object is tested for equality with something else

LOOPING

For statement:

- "for" allows the same operations to be repeatedly preformed on each item in a collection
- syntax :

```
ror <variable > in <collection> :
        <commands using variable>
        < commands using variable>
```

looping fixed # of times :

syntax :

range(<number>)

exclusive

syntax (diffranges or skip values) :

ange (<start> , <stop> , <step>)

step can be negative to reverse direction

vhile statement :

instead of iteration over collection , loop occurs until a condition is untrue

syntax

```
vhile <condition> :
        <commands>
```

break and continue statements :

"break" statement will end loop immediately

"continue" statement will start next iteration of a loop immediately without executing the rest of the current iteration

- both "break" and "continue" statements can be used in either loop statement
- breaking / continuing in a nested loop will only affect the inner most loop your program is currently within]

[Time and complexity :

- total time to run a loop will be length of time of 1 iteration x number of iterations
- in nested loop, the number of iterations of each loop is multiplied then result is multiplied by the length of 1 iteration]

MODULES

[overview:
- script files that store groups of related functions and classes
- keep codes isolated
- to use, most be imported]

[usage:
- 2 methods:
  - "import <module name>"
  - requires prepending objects / functions with the module name

  -OR-

  - "from <modulename> import <object>"
  - requires all but module name

  1. "<modulename> . <object> . <method>"
  2. " <object> . <method> "]

[useful module's:
- argparse * = full featured argument parseng
- csv * = read / wnte CSV files
- smtlib = email (send mail from programs)
- gzip * = directly work with gzip files
- os * = OS related; files, secunty, running programs, etc.
- random = generate random ints, floats, choices, etc
- re * = regular expression processing
- sys = OS related; allows writing to stderr, getting OS, version
- time = time and clock related

* important]

random module :

- random.random() = random floating point between 0-1, exclusive
- random.randint(<start>, <stop>) = random integer , inclusive
- random.shuffle(<collection>) = rearrange elements of an ordered collection into a random order
- random.sample(<collection>, <count>) = extract <count> random items from collection

math module :

- math.ceil(x) = ceiling of x (smallest integer >= x)
- math.floor(x) = floor of x (largest integer <= x)
- abs(x) and math.fabs(x) = absolute value (integer / float)
- math.factorial(x) = factorial
- math.gcd(a,b) = greatest common divisor of a and b
- math.isclose(a, b, rel_tol=1e-09) = checks if a and b are nearly equal
- math.exp(x) = e** x
- math.log(x, base=e) = return log
- math.sqrt(x) = square root

print to stderr :

- sys module lets us send output to stderr
- syntax :

print("<message>", file = sys.stderr)

- default, file parameter is set to "sys.stdout" (standard output)

EXCEPTION HANDLING

overview:

- Python cannot normally handle :
  - syntax errors = grammer that cannot be parsed
  - exceptions = these are ~~system~~ semantic errors particular to the operation
- script will be terminated at the line with error

try / except statement:

- using a " try: " block , code will be monitored for the occurence of an exception during run
- once exception happens , remander of try block is skipped (code with terminal error)
- one "except : " block must follow a try block , indicating what the script should do once exception is encountered.
- if many errors possible, include many "except: " blocks

obtaining exception object:

- an instance object will be created that represents the error and stored as a handle
- can be obtained using " as " keyword

common exception types:

- ValueError = an invalid value has been specified (type conversion)
- TypeError = an invalid data type was specified
- OverflowError = a result was too large to be stored in a variable
- IOError = a file related error occured (reading / writing / not found)

FILE IO

[overview:

- allows python to read / write files directly without needing the user to setup stdin/stdout
- can read/write text and binary
- they use linux security]

[opening files :

- file must be opened using "open" function before read / write
- "open" returns a file object (instance of a file related class)
- "open" requires the mode to open the file
  - "r" = open text file for reading
  - "w" = open text file for writing (existing will be erased)
  - "a" = open text file for appending (will create file if not existant, data will be added to end of existing file)
  - "r+" = open text file for reading and writing
  - "b" = can be added to any mode, treating file as binary]

[reading data :

- three methods :
  - file.read(<N>) = read in N characters. Without N, reads in entire file.
  - file.readline() = read up to and including the next linefeed.
  - file.readlines() = return list of linefeed terminated substring.
- better way to iterate directly on file instance (returns 1 line per iteration)
  - "for line in <file handle> :"
  - where "line" will be set to the current line of each iteration

writing data and position seeking:

- "write" method = "file.write(<string>)" , does not add linefeeds. writing to existing data will overwrite.

- "seek()" method = "file.seek (4)" , with 4 as number that will set current position to 5th byte in file.

with statement:

- opening a file using a "with" statement ensures that the file "close" opperation happens automatically
- file object is now obtained with "as" statement
- "with" statement should be used with IO Error exceptions

OS AND CSV MODULES

[OS file operations:

- current working directory:
  - "os.chdir (<path>)" = change current working directory
  - "os.getcwd ()" = get current working directory

- file and directory manipulation:
  - "os.mkdir (<path>)" : create directory
  - "os.rmdir (<path>)" = remove empty directory
  - "shutil.rmtree (<path>)" = remove non empty directory
  - "os.remove (<path>)" = delete file
  - "os.rename (<src>, <dst>)" = rename file / directory]

[- listing files in directory:
  - "os.scandir (<path>)" = function allows for iteration on each file directory in requested path.
  - "os.walk()" = function can operate recursively

- iteration variable will be set to an instance of os.DirEntry, this contains information about file / directory via attributes / methods:
  - DirEntry.name = filename without path info
  - DirEntry.path = full filepath
  - DirEntry.is_dir() = True if directory is entry
  - DirEntry.is_file() = True if entry is file
  - DirEntry.stat() = returns os.stat_result]

Stat result :

- "os.stat_result" object contains the following attributes :
  - Stat_result.st_mode = mode bits (permissions)
  - Stat_result.st_size = file size
  - stat_result.st_birthtime = creation date
  - Stat_result.st_mtime = modification date

- all dates returned from stat result are expressed in seconds since epoch, to convert use "time.ctime (<time in seconds>)"

reading delimited files :

- "csv" module increases functionality to parse through delimited files.
- "csv" function makes use of a file object , from "open()"
- "csv.reader()" and "csv.DictReader ()" are iterable
- "csv.reader()" returns list of values
- "csv.DictReader()" returns a dictionary where keys are column names
- both take parameters for "delimiter" for changing the delimiter type

writing delimited files :

- "csv.writer()" allows writing a list as delimited text to a file
  - format :

    <variable name> = csv.writer(<list>, delimiter = "<delimiter>", quotechar = "<quotechar>")
- quote character will be used to surround fields that contain delimiter character
- writer.writerow() used to actually write to the file

EXECUTION AND FILE TRANSFERS

[Subprocess module:

- allows python to run programs using "subprocess.run()" method
- function returns an object with access to output
- only use when entirely necessary (better to use python modules than import other commands from bash)
- "subprocess.run(<commands>, shell=True)"
- other possible params:
  - "stdout / stderr" = if set to None (default), run() will not capture and it will be sent to terminal. If set to "subprocess.PIPE", run() will capture
  - "timeout = <int>" = limit execution to specified # of secs.
  - "check = <boolean>" = if true, raises a Called Process Error exception
  - "cwd = <string>" = run program from specified directory.
- module returns an instance of Completed Process class]

[Completed process:

- class that contains the following attributes:
  - args = the command you sent to run()
  - return code = the return code from the program (should be zero)
  - stdout = if set to subprocess.PIPE, this will be text send from progra
  - stderr = same as stdout
- to convert stdout and stderr from binary, use string method "decode("asci")"]

[downloading data and files:

- urllib.request submodule handles HTTP requests
- common methods:
  - urlretrieve() = download a file and directly save

-urlopen( ) = returns instance that can download data from a webpage and be used to save into a Python variable best used in "with" block

° invalid URLs will raise urllib.error.URLError exception]

BIO PYTHON : SEQUENCES

overview:

- open source, bioinformatics package consisting of hundreds of modules / submodules
- areas covered:
  - sequences and associated functionality (complement and translational tables)
  - sequence records (FASTA, GenBank and Alignment records)
  - calling external tools (aligners, MSA, robust BLAST support)
  - Online database access (NCBI entrez queries, Swiss Prot)
  - phylogenetics
  - 3D structure
  - many more (clustering, motifs, pop gen, etc.)

sequences overview:

- "Bio.Seq" module provides sequence functionality
- "Seq" class within Bio.Seq is similar to a Sequence class made by me
- init format:

Bio.Seq.Seq(<sequence>, <alphabet>)

- requires a sequence, alphabet not req
- BioPython supports many alphabets through Bio.Alphabet:
  - Bio.Alphabet.generic_dna = nucleotides, ignore invalid characters
  - Bio.Alphabet.generic_protein = amino acids
  - Bio.Alphabet.IUPAC = methods encountering characters not in the specified IUPAC will raise an exception

[stringlike functionality:

- the Seq class supports most operations you can preform on strings
- Seqs themselves are immutable, just like strings
- these return a Seq, not a string:
  - indexing: my_seq[3]
  - sclicing: my_seq[6:9]
  - concatenation: my_seq1 + my_seq2 (must be compatible alphabets)
- equality can be tested against other Seqs / strings:
  - my_seq1 = = my_seq2
  - my_seq1 = = "AGGCCCTAG"
- the following can use strings / other Seqs as argument:
  - count method = <my_seq>. count ("TT") *non overlapping results*
  - count with overlap = <my_seq>. count_overlap
  - find method = <my_seq>. find ("ATAT")
  - In statement = "TAG" in my_seq
  - starting character = my_seq. startswith ("GATACA")
  - ending with = my_seq. endswith ("CATCATCATCAT")
  - splitting = my_seq. split ("delimiter")] *returns list of Seq instance

[nucleotide methods:

- applicable to nucleotide / unspecified alphabets
- Seq. complement () = preform complement only
- Seq. reverse_complement () = reverse and complement
- Seq. transcribe () = convert T → U
- Seq. back_transcribe () = convert U → T
- Seq. translate () = may be performed on DNA, returns Seq with protein alphabet]

BASIC PYTHON SKILLS

[general if statement:

- format:

"if <condition>:"

- anything within body of statement must begin with indent
- conditional statement must return a True or False Boolean values
- conditional operators:
  - "<operand 1> == <operand 2>" = equality
  - "<operand 1> != <operand 2>" = inequality
  - "<operand 1> > <operand 2>" = greater than
  - "<operand 1> >= <operand 2>" = greater than or equal to
  - "<operand 1> < <operand 2>" = less than
- boolean operator:
  - "<expression 1> and <expression 2>" = AND
  - "<expression 2> or <expression 1>" = OR
  - "not <expression 1>" = NOT
- demorgan's law (boolean):
  - "not (a and b):" = "(not a) or (not b):"
  - "not (a or b):" = "(not a) and (not b):"]

[else statement:

- will run if conditional expression returns False
- syntax:

```
if <condition>:
    <command>
else:
    <command>
```
]

elif statement:

- Short for "else if:", allows checking multiple branches

ternary conditional expressions:

- either one value or another depending on the result of a condition (good for shortening from a multiline if/else)
- "<true value> if <condition> else <false value>"

Jupyter Notebooks:

- tool that allows source code, text markup and inline-output to be distributed as a "notebook"
- supports languages:
  - R
  - Python
  - etc
- behaves like Python's normal script mode
- promotes reproducibility
  - results are saved as part of notebook

Print:

- "print (<message>)" sends text to stdout
- "print" automatically places a line feed after text ("\n" = linefeed)
- "print (<message>, end= "<character>")"
- printing multiple items are separated by a space, using:

"print (<item1>, <item2>, sep = "<delimiter>")"

- changing space to comma makes file CSV compatible
- to insert variable to print:

print ("<message> {0}, {1}... <message>". format(<item0>, <item1>))

ARGPARSE MODULE

[overview:

- contains functionality for parsing command line arguments
- allows for positional, required and optional arguments
- can set default parameters values]

[instantiating ArgumentParser:

- argparse module contains a class, ArgumentParser
- initializer can take the following parameters:
  - "prog" = name of your program
  - "~~destination~~ description" = text thats displayed before arguments
  - "epilog" = text thats displayed after arguments]

[recognized parameters:

- "add_argument" method creates a new parameter the parser will recognize
  - "name(s)" = required argument, will pertain to parameter name ("-<name>" is nonpositional, "<name>" is positional)
  - "help" = description of the parameter
  - "action" = this is the action associated with the parameter
  - "nargs" = the # of arguments associated with the parameter
  - "default" = default value if the parameter isnt specified
  - "required" = if this parameter is required
  - "type" = the data type of the argument
  - "dest" = name of the attribute representing the parameter
- parameter "action":
  - "store" = default
  - "store_const" = stores constant value if param is used
  - "store_true/False" = store True or False
  - "count" = store number of times an argument is given]

[retrieving argument values :

- calling "parse_args()" method will parse all arguments specified by user, store them as attributes in new instance.
- by default, name of instance attribute is same as parameter]

[example :

```
import argparse
```

script

```
parser = argparse.ArgumentParser(description="<what argument provides>")
parser.add_argument("thing", action="store_true", help=<what argument should be>")
parser.add_argument("other", help="this is what other is")

args = parser.parse_args()

print("thing = {0}; other = {1}".format(args.thing, args.other))
```

output

```
thing = True ; other = <what user entered>
```

]

TMUX

[• a window in terminal that will remain open and running even if terminal is not on.]

[Commands for tmux:

- "tmux" = start new session
- " tmux new -s <name>" = to start new session with nam
- "tmux a -t <name>" = open & attach to existing tmux
- " tmux ls" = list tmux windows
- " tmux kill-session -t <name>" = end tmux session]

[Installing tmux:

- in anacondaenvironment, look @ research notes]

[moving a file (server → cluster):

- "scp <file> <PATH> : "