# RELATIONAL ALGEBRA

CYCLE LENGTH:

# Algorithm:
- if even, divide by 2
- if odd, (3*n)+ 1

# Examples:
- c(1) = 1, c(5) = 6, c(10) = 7

COLLATZ:

# Algorithm:
- calculate cycle length
- that is, if 1, stop and return length
- do cycle length calculation if not 1

RMSE:

# Algorithm:
- average the square of the differerences
- take the square root

# Functions:
- numpy, zip_generator_sum, zip_list_sum, map_sum, zip_reduce, zip_for, range_for, while

----

# TESTING

UNIT TESTS:
- self.assertEqual(val1, val2)

ASSERTIONS:
- assert(to_test)
  - to_test must be true
- assert [not] hasattr (var1, "func_name")
  - does var1 have the funcc_name?

EXCEPTION:
- try (asserts inside), except, else, finally
- *YOU CAN "PASS" FOR EXCEPT BLOCK*
- raise excepton_name
- types:
  - IndexError
  - StopIteration

# IMPORTS
- from operator import add, mul, sub, floordiv, truediv
- from math import sqrt
- from numpy import sqrt, mean, square, subtract
- from functools import reduce
- from sys import getrecursionlimit, setrecursionlimit
- from io import StringIO
- to use functions from different class:
  - from class1 import func1
- from itertools import count
- from types import GeneratorType

# LIST COMPREHENSION
- [operation FOR var IN iterable]
- [operation FOR var IN iterable IF what]
- [operation (var1, var2) FOR var1 IN iterable2 FOR var2 IN iterable2]
- calculated before you even call list(L_C)

RECURSSION:
- ***return in base case***
- ***return call function on last line***
- there's a limit on recursion but you can set it to something else; if limit met, runtime error
  - getrecursionlimit ()
  - setrecursionlimit (num1)

# TYPES & OPERATIONS

TYPES:

# Calling "type(x)":
- type(instance_of) = kind of type
- if type(kind of type) = type
  - types ← bool, int, float, complex, str, list, tuple, set, dict, FunctionType,
- EXCEPT FOR CLASSES!
  - type(class_name) = type
  - type(x = class_name) = class_name
    - so instance of class

	List	Tuple
indexable	Yes → assert list[0] == 3	yes
assignment	Yes → list[0] += 1 → list[0] = 5	No
replication	Yes → list = [1,2] list2 = 2 * [1,2] then list2 = [1,2,1,2]	Yes
length	Yes → len(list)	Yes
immutable	No → can't add values in existing tuple; must make new instance	Yes

----

OPERATORS:
- **addition:** (+)
  - add(num1, num2)
  - num1 += num2
- **true division:** (/)
  - returns float
  - 5 / 2 = 2.5
- **floor division:** (//)
  - returns int if both ints, float if one or more float
  - 5 / 2 = 2
- **mod:** (%)
  - remainder
- **replication:** (*)
  - doubles str, list, tuple
- **exponent:** (**)
- **bit shift left:** (<<)
- **bit complement:** (~)
- **bit and:** (&)
- **bit or:** (|)
- **bit xor:** (^)
  - if both 0 = 0
  - otherwise 1

# ITERABLES
- always indexable
- have __iter__
- doesn't have __next__
- **SETS:**
  - order not gaurenteed
- **DICTS:**
  - { 'key1' : val1, 'key2' : val2 }
  - set(dictName.keys())
  - set (dictName.values())
  - set (dictName.items())
    - return set of tuples
    - so you can iterate through (for k,v in dictName.items())
- **RANGE:**
  - range(10) = 0-9
  - range (2, 10) = 2-9
  - range (0, 10, 2) = 0, 2,…8
  - range (10, 0, -2) = 10, 8,…2
  - has __getitem__ function
  - IndexError when out of range
  - can't set values (i.e. r[0] = 3)

# ITERATORS
- consumed after one use
  - you can't call y again EVER
- both __next__ and __iter__
- call iter() on an iterator and get itself
- **COUNT:**
  - not indexable
  - count(0) = 0,1,2…
  - count(3,2) = 3,5,7…
- **GENERATORS:**
  - Call Looks like list comprehension with ()
    - y = (blah)
  - Only happens after you call y
    - i.e. list(y)
  - yield in function makes it generator
  - call next (var) to go through
- **MAP:**
  - map (function, iterable)
  - applies function to every element in iterable
  - only happens after you call

# FUNCTIONS
- **FORMAT:**
  - def f () :
  - def f (self) :

CONSUMED AFTER ONE CALL/USE:
- **SORT:**
  - sorted (iterable)
- **REVERSED:**
  - reversed (iterable)
- **ENUMERATE:**
  - enumerate (iterable)
  - returns tuples in casing with whole numbers as first element and iterable value as second

RANDOM ONES:
- **MAP:**
  - make function apply to every iterable
- **LAMBDA:**
  - anonymous functions; not bound to a name at runtime
  - you can use it for map, reduce, etc.
  - lambda x : x **2
- **REDUCE:**
  - reduce(function, iterable)
  - apply function from left to right to get one single value
  - reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
  - x is the accumultated value on
  - y is the current value of the iteratable
- **ZIP:**
  - Returns iterator of tuples
  - Basically takes an element from each iterable and puts it into each tuple
  - Iterables don't have to be equal length

# FUNCTION CALLS/PARAMS

- must have exact amount of references to param as num of param
- function names used in call must match names in function's param
- call by name is after position
    - error ex: f(a=1, 3, 7)
- no multiple calls to same param
    - error ex: f(a=1, b=3, a=3)
- ORDER: position, call by name (a=3), unpacking
- ERRORS:
    - Typeerror if names don't match
    - Syntaxerror if order incorrect

"*" IN A PARAMETER LIST (FUNCTION TUPLE)

- packs up into a tuple
- i.e. f(x, y)
- f (1,2,3) = [1, (2,3)]
- tuple can end up being empty or single value or both filled up

"**" IN A PARAMETER LIST

- packs into dictionary
- must speficiy in call the key and the value by name
    - f (2, 3, a = 4)

"=" IN A PARAMETER LIST (DEFAULT)

- default values
- when calling function, don't have to assign the defaulted params but If you do, override the default
- def f (x,y,z=5)

"*" IN A FUNCTION CALL (UNPACKING)

- unpacks values and uses them for arguments
- for example, f (a, b*) = f (a, b1, b2)

"**" IN A FUNCTION CALL

- gets values out of dictionary using it by name
- keys must match param names
- purpose: you can't pass dictionaries

"=" IN A FUNCTION CALL

- just setting it

# RELATIONAL ALGEBRA

SELECT:

- select rows that match criteria/function
- params: iterables, function

# functions:

- **yield:** use typical for and if
- **generator:** use list comprehension

PROJECT:

- returns columns of iterables
- function call: iterables, n column values
- **function:** iterables, packed tuple

# functions:

- **yield:** for and list comprehension
- **generator:** same as yield but move outer for loop to outer in list comprehension to make it one line

CROSS JOIN:

- takes two sets of iterables and does a cross join (all combinations)
- params: iterables1, iterables2

# Algorithm:

- yield: use nested for loops and dict
- generator: same thing but one line with list comprehension

# Dict:

- class dict(**kwarg)
- class dict(mapping, **kwarg)
- class dict(iterable, **kwarg)

THETA JOIN:

- only join if function is true
- params: iterables1, iterables2, func

# Algorithm:

- yield & generator: same as cross join. Just add an if for the func.

NATURAL JOIN:

- only join if all same keys have same values
- params: iterables1, iterables2

# Algorithm:

- yield: check if iterators are equal but values

# SQL

ORDER:

- select, from, where, order by

# Drop: deletes table
# drop, create, insert, show,
# select, project, cross join, theta join, natural join

# MISC.

COMPOSITION:

DECORATORS:

INHERITENCE:

REFLECTION:

# REFACTORING

EXTRACT METHOD:

- **LOOK FOR CHUNKS OF CODE!**
- You have a code fragment that can be grouped together
- To-Do: Turn the fragment into a method whose name explains the purpose of the method.
- Purpose: easier to read; reads like comments

MOVE METHOD:

- **LOOK FOR METHODS THAT USE OTHER CLASS OBJECTS**
- Method is using (or will use) more features in a different class than the one that it was defined in
- To-Do: either create a new, similar method in the other class or move the method to the other class
- Purpose: make classes simpler; less coupling

REPLACE TEMP WITH QUERY:

- **LOOK FOR TEMP VARIABLE WITH EXPRESSION; REPEATEDLY CALLED**
- You are using a temporary variable to hold the result of an expression
- To-Do: Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods; CALL THAT NEW METHOD every time you need the expression.

REPLACE TYPE CODE WITH SUBCLASSES:

- **LOOK FOR THE "TYPE" VARIABLE**
    - If type value changes after creation or subclasses already exist, use state/strategy
- You have an immutable type (*tuple, str, int/long, float, complex, bool)* code that affects the behavior of a class.
- To-Do: Replace the type code with subclasses.
- Signatures:
    - class Subclass (BaseClass)
    - __init__ method

REPLACE TYPE CODE WITH STATE/STATEGY:

- You have a type code that affects the behavior of a class, but you cannot use subclassing.
- To-Do: Replace the type code with a state object.

REPLACE CONDITIONAL WITH POLYMORPHISM:

- YOU NEED replace type code with subclasses or replace type code with state/strategy first!
- You have a conditional that chooses different behavior depending on the type of an object
- To-Do: Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.
    - Extract method on conditional
    - Move method to make sure conditional is at top of inheritance (base class)
    - Create subclass method of one subclass tha overrides conditional
- Purpose: allows you to avoid writing an explicit conditional when you have objects whose behavior varies depending on their types

----

# DON'T BE STUID

1) YOU CAN'T PASS DICTIONARY IN FUNCTION CALL! MUST UNPACK IT!

****IMMUTABLE****

tuple, str, int/long, float, complex, bool

ADDING NEW VALUE TO ITERABLE:

- add iterable (set or tuple, etc.) to value at index
- *YOU'RE NOT MODIFIYING THE VALUE ITSELF IF IT'S IMMUTABLE*
- *You can make a new tuple with an edit (i.e. string concatenation) but you're not modifying the original*