Arrays: array, definition

A Python array is an object type which can compactly represent an array of basic values: characters, integers, and floats, Though they behave mostly like lists, they can only store the type declared.

Arrays: Keep array sorted with insort

```
from array import array
from bisect import insort

a = array('l', sorted([3,2,9,5,10,-1]))
insort(a, 7)
a  # array('l', [-1, 2, 3, 5, 7, 9, 10])
```

Arrays: Type codes

```
from array import array

my_array = array('?')

# replace ? with a type code:

b signed 1-byte integer
B unsigned 1-byte integer
u Unicode character
l signed 4-byte integer
L unsigned 4-byte integer
f floating point (4 bytes)
d floating point (8 bytes)
```

Bits: Create byte sequence from hex values

```
b = bytes.fromhex("31 4b ce a9")  # b'1K\xce\xa9'
```

Bits: Find the single number among pairs of numbers

Given an array where there is a single unique number and every other number occurs twice, find this single number.

Trick: an XOR operation will set the first number, and is reversible. When numbers overlap, the latest operation will be reversed when the number is encountered again.

Note: XOR is commutative, a ^ b == b ^ a, and associative, a ^ (b ^ c) == (a ^ b) ^ c.

```
def single_integer(a):
    runningVal = 0
    for i in a:
        runningVal ^= i
    return runningVal
```

Bits: Invert ones and zeros

```
# use the NOT operator, ~ (tilde)

x = 0b10011
n = ~x
print(x | n)  # -1 (see Twos' complement)
```

Bits: Representing numbers in binary

```
# decimal to binary
n = 30
s = bin(n)[2:]  # '11110'

# binary to decimal
x = int(s, 2)   # int(string, base)
testeql(n, x)
```

Bits: Toggle a bit

Toggle a bit at index idx, swapping a 0 with a 1 and a 1 with a 0.

```
def toggle_bit(n, idx):
    """Toggle bit at index idx,
    where the rightmost bit is 0,
    then the second-to-last is 1"""

    toggle_mask = 1 << idx
    return n ^ toggle_mask  # use XOR
```

Bits: Turning individual bits on or off

Given an arbitrary bit sequence, turn a specific bit on or off

```
def turn_bit_on(n, idx):
    """Turn n's bit on (set to 1) at index idx,
    where the rightmost bit is 0,
    then the second-to-last is 1"""

    # idea:
    #    0b?????
    # OR 0b00100
    #    -------
    #        x   will always equal 1

    on_seq = 1 << idx
    return n | on_seq

def turn_bit_off(n, idx):
    """Turn n's bit off (set to 0) at index idx,
    where the rightmost bit is 0,
    then the second-to-last is 1"""

    # idea:
    #     0b?????
    # AND 0b11011
    #     -------
    #         x   will always equal 0

    bit_len = n.bit_length()
    off_seq = ~(1 << idx)  # use NOT operator

    return n & off_seq
```

Bits: Twos' complement

See a negative value as a series of bits without the minus sign

```
n = -12
width = 16
bin(n + (1 << width))
```

Booleans: all(), any()

```
# no need to make a list inside all or any (with [ ])

array = [1,2,3,4,5]
all(value > 2 for value in array)

any(value % 2 == 1 for value in array)
```

Calculus: Differentiation

Given an array of coefficients, where the rightmost element is a constant, second-to-last is x, third-to-last is x^2, etc. find the derivative

```
def differentiate(coeffs):
    """Differentiate 2x^3 - 4x^2 + 10 (given as [2, -4, 0, 10]).
    The result is 6x^2 - 8x. [6, -8, 0]
    """
    len_terms = len(coeffs)
    result = []
    for i, coeff in enumerate(coeffs[:-1]):
        power = len_terms - i - 1
        result.append(power * coeff)
    return result
```

Calculus: Integration, definite

```
def definite_integral(coeffs, a, b):
    """For the integral of a polynomial with given coefficients,
    where the rightmost element is a constant, second-to-last is x,
    third-to-last is x^2, etc.

    For definite_integral([1, 0, 1], 0, 2),
    the anti-derivative is (1/3)x^3 + x = [1/3, 0, 1]
    """
    anti_derivative = []  # ignore C, constant value
    len_coeffs = len(coeffs)
    for i, coeff in enumerate(coeffs):
        power = len_coeffs - i
        anti_derivative.append(1 / power * coeff)

    # compute anti_derivative with a and b
    len_anti_derivative = len(anti_derivative)
    result = 0
    for i, coeff in enumerate(anti_derivative):
        power = len_anti_derivative - i
        result += coeff * b ** power
        result -= coeff * a ** power
    return result
```

Combinatorics: Combinations with bit strings

A combination can be thought of a mapping between a bit string and the indices of the whole collection. If an index is 1 (True), the element is in the combination; 0 (False) means it is not. We iterate through bit strings with n ones (making it have n elements).

```
def push_rightmost_bit(bs):
    """Find next bit string with the same number of ones"""
    rightmost_10 = bs.rfind("10")
    if rightmost_10 == -1:
        return False  # cannot push any more
    # count number of ones to the right of 10
    ones_to_move = bs[rightmost_10 + 1:].count("1")
    zeros_left = len(bs) - rightmost_10 - ones_to_move - 2
    return bs[:rightmost_10] + "01" + ("1" * ones_to_move) + ("0" * zeros_left)

def bitstring_combination(a, bs):
    return [elem for (i, elem) in enumerate(a) if bs[i] == "1"]
```

```
def my_combinations(a, n):
    a = list(a)
    len_a = len(a)
    bs = ('1' * n) + ('0' * (len_a - n))
    out = []
    while bs:
        out.append(bitstring_combination(a, bs))
        bs = push_rightmost_bit(bs)
    return out
```

Combinatorics: Combinations, generating

itertools.combinations(iter, n) produces subsequences of length n.
combinations_with_replacements allows elements to be repeated as many times as possible.

```
import itertools
list(itertools.combinations('cba',2))
# [('c', 'b'), ('c', 'a'), ('b', 'a')]

[''.join(c) for c in itertools.combinations_with_replacement('cba', 3)]
# ['ccc', 'ccb', 'cca', 'cbb', 'cba', 'caa', 'bbb', 'bba', 'baa', 'aaa']
```

Combinatorics: Combinations, number of

The number of r-combinations (combinations with r elements) of a set with n elements is: n! / (r! * (n - r)!)

```
# r-combinations out of a list of n elements
n! / (r! * (n - r)!)
```

Combinatorics: Permutations, generating

permutations(iter, n=len(iter)) produces permutations of the items in iter of length n.

```
import itertools
list(itertools.permutations('abc', 2))
# [('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('c', 'a'), ('c', 'b')]
```

Combinatorics: Permutations, iterative method

An iterative method to produce permutations of length n from a sequence a. The idea is to branch out at each step with all possible single items left from the remaining items, accumulating to each result.

```
def permutation_branch(pair):
    """Given (base, rest), take each item in rest and append to base)"""
    out = []
    base, rest = pair
    for i, n in enumerate(rest):
        out.append((base + [n], rest[:i] + rest[i+1:]))
    return out

def my_permutations(a, n):
    a = list(a)
    out = [([], a)]

    for i in range(n):
        step = []
        for pair in out:
            step.extend(permutation_branch(pair))
        out = step
    return [p for (p, rest) in out]
```

Combinatorics: Permutations, number of

The number of r-permutations (permutations with r elements) of a set with n elements is: n! / (n - r)!. For example, to choose 3 winners out of 100, we have 100 people for first place, 99 for second, and 98 for third = 100 * 99 * 98. (100 - 3)! = 97!, which are the values cancelled outfrom 100!

```
# r-permutations out of a list of n elements
n! / (n - r)!
```

Combinatorics: Product (Cartesian) combines elements of multiple lists

itertools.product(it1, it2, ... itn, repeat=1) produces tuples of n elements, where n are the number of given iterables. repeat indicates how many times the given iterables are repeated.

```
import itertools

[''.join(p) for p in itertools.product("AB", "CD", "EF")]
# ['ACE', 'ACF', 'ADE', 'ADF', 'BCE', 'BCF', 'BDE', 'BDF']

[''.join(p) for p in itertools.product("AB", repeat=3)]
# ['AAA', 'AAB', 'ABA', 'ABB', 'BAA', 'BAB', 'BBA', 'BBB']
```

Concurrency: Concurrency and parallelism

Concurrency is about structure, and parallelism is about execution. Concurrency produces a way to structure a solution to a problem that may be (but not necessarily) parallelizable.

Datetime: Format date with strftime

The mnemonic for strftime and strptime is that 'f' stands for format, and 'p' stands for parse

```
from datetime import date, datetime
dt = datetime(2017, 10, 15, 23, 22)
d = date(2017, 2, 15)

print(dt.strftime("%H:%M %d/%m/%y"))  # 23:22 15/10/17
print(d.strftime("%d/%m/%y"))  # 15/02/17
```

Datetime: now

Note: date does not have now()

```
from datetime import datetime

now = datetime.now()
# datetime.datetime(2017, 10, 30, 19, 50, 29, 702774)
```

Datetime: Parse date with strptime

The mnemonic for strftime and strptime is that 'f' stands for format, and 'p' stands for parse. strptime is a class method of datetime.datetime.

```
from datetime import datetime
d = datetime.strptime("02/15/97", "%m/%d/%y")
# datetime.datetime(1997, 2, 15, 0, 0)
```

Datetime: Weekdays

weekday() may be called by both a date and a datetime.

0 = Mon, 1 = Tue, 2 = Wed, 3 = Thu, 4 = Fri, 5 = Sat, 6 = Sun

Decorators: Copy attributes of decorated function to decorator

To avoid __name__ and __doc__ being masked by a decorator, use `@functools.wraps`

```
import functools

def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        # code omitted
```

Decorators: Decorators with arguments

To have a decorator that accepts arguments, we must create a decorator factory that accepts arguments and returns a decorator.

```
registry = set()

def register(active=True):
    def decorate(func):
        print('running register(active=%s) -> decorate(%s)'
              % (active, func))
        if active:
            registry.add(func)
        else:
            registry.discard(func)
        return func
    return decorate

@register(active=False)  # same as register(active=False)(f1)
def f1():
    print('running f1()')
```

Decorators: Decorators, definition

A decorator is a callable that accepts another function as an argument (the decorated function).

A decorator can do some processing with the decorated function and return it or substitute it with another function.

Decorators are executed immediately after the decorated function is defined (typically when the module is imported, called import time)

```
@deco
def target():
    print("Running target()")

# is the same as

def target():
    print("Running target()")

target = deco(target)
```

Decorators: Register functions in a registry

```
registry = []
def register(func):
    print("Running register(%s)" % func)
    registry.append(func)
    return func

@register
def f1():
```

```
    print("Running f1()");

@register
def f2():
    print("Running f2()");
```

Decorators: Stacking decorators

Nesting decorators work inside out

```
@d1
@d2
def f():
    return "f"

# is the same as

f = d1(d2(f))
```

Decorators: Substituting the decorated function

```
def deco(func):
    def inner():
        print("Running inner()")
    return inner

@deco
def target():
    print("Running target()")

target()  # Running inner()
```

Dictionaries: Counter

A dict subclass for counting hashable items. Also called a bag or multiset

```
from collections import Counter

s = "abracadabra"
s_ctr = Counter(s)  # Counter({'a': 5, 'b': 2, 'r': 2, 'd': 1, 'c': 1})

p = "panama"
p_ctr = Counter(p)

s_ctr - p_ctr  # Counter({'a': 2, 'b': 2, 'r': 2, 'd': 1, 'c': 1})

# total of all counts
sum(s_ctr.values())  # 11

s_ctr.values()  # dict_values([2, 1, 2, 5, 1])
s_ctr.keys()    # dict_keys(['b', 'd', 'r', 'a', 'c'])
```

Dictionaries: Creating a dict

```
a = dict(a=1, b=2)
b = {'a': 1, 'b': 2}
c = dict(zip(['a', 'b'], [1, 2]))
d = dict([('a', 1), ('b', 2)])
e = dict({'a': 1, 'b': 2})
```

Dictionaries: defaultdict

Values are created on-demand when a missing key is searched. A function or class is passed as the argument to defaultdict

Note: dd.get(missing_key) returns None

```
from collections import defaultdict

dd = defaultdict(list)
dd['a'] = [10]
dd['c'] = [30]

letters = 'abcde'
for i, letter in enumerate(letters):
    dd[letter].append(i+1)

dd  # defaultdict(<class 'list'>,
  # {'e': [5], 'c': [30, 3], 'a': [10, 1],
  #  'd': [4], 'b': [2]})
```

Dictionaries: dictcomp (dict comprehension)

```
DIAL_CODES = [
        (86, 'China'),
        (91, 'India'),
]

cc = {country: code for code, country in DIAL_CODES}
```

Dictionaries: Get default value if key not found

```
d = {'a': 1, 'b': 2}
c = d.get('c', 99)
```

Dictionaries: Invert keys and values

Assume it is one-to-one. Otherwise a single, random key will become the value in the inverted dict.

```
d = {'a': 1, 'b': 2, 'c': 30, 'd': 400}

d_inv = {v: k for k, v in d.items()}
#   {400: 'd', 1: 'a', 2: 'b', 30: 'c'}
```

Dictionaries: namedtuple

A lightweight, dictionary-like object. Fields are accessed with dot notation.

```
from collections import namedtuple

Card = namedtuple("CardDisplay", 'rank suit')
# alternatively namedtuple("CardDisplay", ['rank', 'suit'])

big_two = Card(2, 'Spades')
print(big_two.suit)

print(big_two._asdict())
```

Dictionaries: OrderedDict

Maintain keys in the order of insertion. popitem(last=True) removes, by default, last-in, first-out.

```
from collections import OrderedDict
from operator import itemgetter

d = {'a': 1, 'b': 2, 'c': 3}
od = OrderedDict(sorted(d.items(), key=itemgetter(0)))
```

Dictionaries: Set to default value if key not found

If the key, k, is in dict d, return d[k]. Otherwise, set d[k] = default value

```
d = {'a': 1, 'b': 2}
b = d.setdefault('b', 99)
c = d.setdefault('c', 99)
b, c, d  # (2, 99, {'c': 99, 'a': 1, 'b': 2})
```

Dictionaries: Sort by value

```
d = {'a': 3, 'b': 1, 'c': 2}
sorted(d, key=d.get)  # ['b', 'c', 'a']
```

Dictionaries: Subclass UserDict instead of dict

```
from collections import UserDict

class StrKeyDict(UserDict):
    pass
```

Dictionaries: Update a dictionary

```
d = {'a': 1, 'b': 2}
w = {'z': 26}

d.update(w)  # {'z': 26, 'a': 1, 'b': 2}

u = [('c', 3), ('d', 4)]  # or tuple of tuples
d.update(u)  # {'z': 26, 'd': 4, 'a': 1, 'b': 2, 'c': 3}
```

Dynamic Programming: Coin counting

Given a list of N coins valued (V1, V2, V3, ...) and a total sum S, find the minimum number of coins to total S (we can use as many coins of one type as we want), or report that it is not possible to get to sum S

```
def minCoins(coins, s):
    dp = [float('inf')] * (s+1)
    dp[0] = 0

    for i in range(s+1):
        for j in range(len(coins)):
            if coins[j] <= i and dp[i-coins[j]] + 1 < dp[i]:
                dp[i] = dp[i-coins[j]] + 1
    return dp[-1]
```

Dynamic Programming: Memoization

Memoization is the process of storing previously computed results in a quickly accessible place (a dictionary). The built-in `@functools.lru_cache` (Least Recently Used Cache) is an easy way to memoize a function.

The signature is: `@functools.lru_cache(maxsize=128, typed=False)`

If maxsize is None, the cache can grow without bound. If typed is True, arguments of different types will be cached separately.

```
import functools

@functools.lru_cache()
def fibonacci(n):
    if n < 2:
```

```
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Dynamic Programming: Rod cutting

A rod of length L is to be cut into several pieces, or left intact. Given an array of prices for pieces of increasing lengths, determine the most money that can be made from this rod.

```
def cutRodBottomUp(prices, length):
    prices = [0] + prices
    r = [0] * (length+1)
    r[0] = 0
    for j in range(1, length+1):
        q = float('-inf')
        for i in range(1, j+1):
            q = max(q, prices[i] + r[j-i])
        r[j] = q
    return r[length]
```

Dynamic Programming: Top-down vs. Bottom-up

Using a top-down approach, the procedure is written recursively in a natural matter, but also uses memoization to save the result of each subproblem.

A bottom-up approach typically depends on some natural notion of the "size" of a subproblem, such that solving a subproblem depends only on solving "smaller" subproblems. Subproblems are sorted by size and solved in order, smallest first

Emacs Setup: .emacs

Add to your .emacs file

```
(global-set-key (kbd "<f5>") 'run-python)

(setenv "PYTHONPATH" "/home/heitor/shared/python/my-modules/")

(setenv "PYTHONSTARTUP" "/home/heitor/shared/python/my-startup.py")

(defun my-python-test-buffer ()
  (interactive)
  (save-excursion
    (python-shell-send-string (trim-string (buffer-string)))
    (python-shell-send-string (concat "print('')\n" "test()"))))

(add-hook 'python-mode-hook
          (lambda ()
            (local-set-key [M-return] 'my-python-test-buffer)))

(add-hook 'inferior-python-mode-hook
          (lambda ()
            (auto-complete-mode 1)))
```

Emacs Setup: Python mytests

```
# file: mytests.py

import sys

def testequal(expression, expected):
    print("testing", expression, "expecting", expected)

    # special case: floats, check up to 6 decimal places
    if isinstance(expression, float):
        test_passed = abs(expression - expected) < 1e-6
```

```
    else:
        test_passed = expression == expected
    if test_passed:
        print("(^o^) PASS\n")
    else:
        print("(>_<) FAIL\n")

# alias
testeql = testequal

def pr(s):
    """pr('a b c') prints each of the names separated by a space"""
    if type(s) != str:
        raise ValueError("Argument to pr() must be a string")
    frame = sys._getframe(1)
    names = s.split()
    for name in names:
        print(name, '=', repr(eval(name, frame.f_globals, frame.f_locals)), end=", ")
    print()
```

Emacs Setup: Python Startup

```
# file: my-startup.py

from math import exp, log, sin, cos, tan, asin, acos, atan, floor, ceil
import math
import re
from mytests import testeql, pr

# add common strings to include in autocomplete

print("testeql")
```

Functions: A function may alter mutable arguments

```
def f(a, b):
    a += b
    return a

x = 1
y = 2
f(x, y)  # 3, but x is not altered

a = [1, 2]
b = [3, 4]
f(a, b)  # a becomes [1, 2, 3, 4]

t = (10, 20)
u = (30, 40)
f(t, u)  # (10, 20, 30, 40), but t is not altered
```

Functions: applying a function to dynamic list of arguments

`apply` was removed because a function can be called with starred arguments

```
# instead of apply(fn, args, kwargs), call

fn(*args, **kwargs)
```

Functions: Avoid mutable parameters as default values

The default value is evaluated when the function is defined, so when the mutable object is changed, this change will affect all future calls to that function. The solution is to use None as the default value, and make a copy of the argument passed.

```
class Bus:
    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []
        else:
            self.passengers = list(passengers)
```

Functions: Check if an object is callable

```
callable(2)  # False
callable(lambda x: x+1)  # True
callable(callable)  # True
```

Functions: Closures

A closure is a function that has access to existing free variables when the function is defined, so that they may be used later when the scope of the definition is no longer available

```
# calculate the average of a series of values
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)  # here, series is a free variable
        total = sum(series)
        return total / len(series)

    return averager

# inefficient because the sum is repeatedly computed

# nonlocal is not needed because we were not assigning to series,
# only calling append (lists are mutable)

# a better solution uses 'count' and 'total',
# with nonlocal in averager

def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal count, total
        count += 1
        total += new_value
        return total / count

    return averager
```

Functions: Estimate from midpoints

Given a target, update guesses until the desired value is within a tolerance

```
def sqrtEstimateMidpoint(x):
    left = 0
    right = x

    while True:
        midpoint = (right + left) / 2
        squareMidpoint = midpoint * midpoint

        if abs(squareMidpoint - x) < 1e-4:
            return midpoint

        if squareMidpoint > x:
            right = midpoint
```

```
        else:
            left = midpoint
```

Functions: filter

If predicate is True, the value is kept. itertools.filterfalse keeps items for which the predicate is False

```
def is_odd(n):
    return n % 2 == 1

list(filter(is_odd, range(10)))  # [1, 3, 5, 7, 9]

list(filter(lambda x: x > 2, range(6)))  # [3, 4, 5]

# alternative way, with listcomp
[n for n in range(10) if n % 2 == 1]
```

Functions: First-class function, definition

A first-class function:

- can be created at runtime
- can be assigned to a variable or inside a data structure
- can be passed as an argument to a function
- can be returned as the result of a function call

Functions: Fixed point, finding

Find x such that f(x) = x

```
def fixedPoint(f, x):
    prev = x
    trial = f(x)
    while abs(trial - prev) > 0.0001:
        prev = trial
        trial = f(trial)
    return trial

def fixedSqrt(x):
    return fixedPoint(lambda y: 0.5 * (x/y + y), x)

fixedSqrt(2)  # 1.414213562...
```

Functions: Higher-order function, definition

A higher-order function is a function that accepts a function as an argument or returns a function.

Functions: lambda creates anonymous functions

Lambdas are generally used in higher-order functions. In the example, we sort by the word ending to look for rhymes

```
fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
sorted(fruits, key=lambda word: word[::-1])
```

Functions: map, starmap

map applies the given function to each element of the given iterable. starmap(f, iter) returns an iterable that applies f(*item_iter) for each item_iter that iter produces

```
def square(x):
    return x * x
```

```
a = [1, 2, 3, 4, 5]
list(map(square, a))  # [1, 4, 9, 16, 25]

# alternative way, with listcomp
[square(n) for n in a]

# starmap
import itertools, operator
list(itertools.starmap(operator.mul, enumerate('abc', 1)))
# ['a', 'bb', 'ccc']
```

Functions: methodcaller

```
from operator import methodcaller

hyphenate = methodcaller('replace', ' ', '-')
hyphenate("something to do")  # 'something-to-do'
```

Functions: Newton's method to approximate an equation's solution

If x->g(x) is a differentiable function, then a solution of the equation g(x) = 0 is a fixed point of the function x->f(x), where f(x) = x - (g(x) / Dg(x)), where Dg(x) is the derivative of g evaluated at x.

A number x is a fixed point of a function f if f(x) = x. For some functions, repeatedly applying f(x), f(f(x)), f(f(f(x)))... can be done to find x.

Functions: partial application

`partial` freezes part of the arguments passed to a function. By default, only the leftmost arguments may be frozen.

`partialmethod` works for methods.

```
from unicodedata import normalize
from functools import partial

# typically, we would call normalize('NFC', s)
nfc = partial(normalize, 'NFC')
nfc('café')
```

Functions: Recursion, basic (Fibonacci sequence)

Fibonacci sequence, not optimal but conceptually easy to understand.

```
def fibRec(n):
    """Return the nth Fibonacci number. fib(0) = 0 and fib(1) = 1"""
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fibRec(n-1) + fibRec(n-2)

def test():
    """0, 1, 1, 2, 3, 5, 8"""
    testeql(fibRec(6), 8)
```

Functions: reduce

reduce applies the given function to items of the iterable successively, returning the accumulated result

```
from functools import reduce
from operator import mul

reduce(mul, range(1, 7))  # 720, same as factorial(6)
```

Functions: Rich comparisons, autocompleting

The decorator `@functools.total_ordering` supplies the class' remaining rich comparison ordering methods when one or more are of them defined.

```
@total_ordering
class Student:
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

Functions: Single-dispatch

Decorating a simple function with `@functools.singledispatch` makes it a generic function (a group of functions that behaves in different ways, depending on the type of the first argument)

```
from functools import singledispatch
import numbers
import html

@singledispatch
def htmlize(obj):
    content = html.escape(repr(obj))
    return "<pre>{}</pre>".format(content)

@htmlize.register(str)  # specify the type of the argument
def _(text):  # the name of a specific function is irrelevant; use _
    content = html.escape(text).replace('
', '<br>
')
    return "<p>{}</p>".format(content)

@htmlize.register(numbers.Integral)
def _(n):
    return "<pre>{0} (0x{0:x})</pre>".format(n)
```

Functions: Variable number of arguments

```
def describe(category, *items, **properties):
    print(category)
    print(' '.join(items))
    print('
'.join("%s : %s" % (key, val) for key, val in properties.items()))

describe("Games", 'Poker', 'Blackjack', 'Chess', owner="Joe", winner="Tim")
```

Geometry: Degrees to radians

```
from math import pi
```

```
def degToRad(d):
    return d / 180 * pi
```

Geometry: Law of cosines

The law of cosines relates the lengths of a triangle's sides to the cosine of one of its angles.

```
from math import cos, sqrt

def sideLen(a, b, angleC):
    return sqrt(a**2 + b**2 - 2*a*b*cos(angleC))
```

Geometry: Radians to degrees

```
from math import pi

def radToDeg(r):
    return r / pi * 180
```

Graphs: BFS (Breadth-First Search)

```
from collections import deque

def bfsConnectedComponent(m, start):
    """Given an adjacency matrix and starting node, return the set
    of vertices connected to the starting node"""

    q = deque([start])  # queue, use append and popleft
    visited = {start}  # set
    component = []  # output

    while len(q) > 0:
        cur = q.popleft()
        component.append(cur)

        for vertex, connected in enumerate(m[cur]):
            # vertex is the column index in matrix (i)
            # connected is the True/False, 1 or 0 value
            if connected and not vertex in visited:
                q.append(vertex)
                visited.add(vertex)
    return component
```

Graphs: Detect a cycle in an undirected graph

```
def isCyclic(m):
    visited = set()
    found = False

    def isCyclicStep(v, parent):
        nonlocal found

        if found:
            return
        visited.add(v)
        for vertex, connected in enumerate(m[v]):
            if connected and vertex in visited and vertex != parent:
                found = True
            if connected and not vertex in visited:
                isCyclicStep(vertex, v)

    for i in range(len(m)):
        if not i in visited:
            isCyclicStep(i, i)
        if found:
```

```
            break
    return found
```

Graphs: DFS (Depth-First Search), iterative

The difference between DFS and BFS is that DFS uses a stack (vertices encountered last are processed first), while BFS uses a queue (vertices encountered first as immediate neighbors are processed first)

```
def dfsIterative(m, start):
    """Given an adjacency matrix and starting node,
    traverse the graph"""

    s = [start]  # list, use as stack
    visited = {start}  # set
    out = []

    while len(s) > 0:
        cur = s.pop()
        out.append(cur)

        for vertex, connected in enumerate(m[cur]):
            # vertex is column in matrix (i)
            # connected is the True/False, 1 or 0 value
            if connected and not vertex in visited:
                s.append(vertex)
                visited.add(vertex)
    return out
```

Graphs: DFS (Depth-First Search), recursive

```
def dfsRecursive(m, start):
    visited = set()
    out = []
    def dfsRecursiveStep(start):
        visited.add(start)
        out.append(start)
        for vertex, connected in enumerate(m[start]):
            # vertex is column in matrix (i)
            # connected is the True/False, 1 or 0 value
            if connected and not vertex in visited:
                dfsRecursiveStep(vertex)
    dfsRecursiveStep(start)
    return out
```

Graphs: Representation, Adjacency List

An adjacency list may be a list of lists or a dictionary, where the source (initial) vertex is the key and sink (terminal) vertices are stored in a corresponding list.

```
adjLists = [[1, 2], [2, 3], [4], [4, 5], [5], []]

adjListsDict = {}
adjListsDict[0] = [1, 2]
adjListsDict[1] = [3]
```

Graphs: Representation, Adjacency Matrix

The adjacency matrix is an n x n matrix with True or False values, where True represents an edge between the row-numbered vertex and the column-numbered vertex. An undirected graph is symmetric.

In the example below, the first line is the number of vertices and each subsequent line, an edge.

```
GRAPH = """
3
0 1
1 2
2 0
"""

def buildAdjMatrix(s):
    lines = filter(lambda line: len(line.strip()) > 0, s.split("
"))
    nodes = int(next(lines))
    m = [[0 for _ in range(nodes)] for _ in range(nodes)]
    for edgeStr in lines:
        ends = list(map(int, edgeStr.split()))
        m[ends[0]][ends[1]] = m[ends[1]][ends[0]] = 1
    return m
```

Heaps: Heap data structure

A heap may be stored in an array or a binary tree.

Heaps: Heapsort

```
class Heap():
    def __init__(self, unsorted):
        self.a = unsorted
        self.heapSize = len(unsorted)

    # consider this structure
    #      0
    #     / \
    #    1   2
    #   / \ / \
    #  3  4 5  6

    def left(self, i):
        """index of left child"""
        return 2*i + 1

    def right(self, i):
        """index of right child"""
        return 2*i + 2

    def maxHeapify(self, i):
        left = self.left(i)
        right = self.right(i)
        if left < self.heapSize and self.a[left] > self.a[i]:
            largest = left
        else:
            largest = i
        if right < self.heapSize and self.a[right] > self.a[largest]:
            largest = right
        if largest != i:
            self.a[i], self.a[largest] = self.a[largest], self.a[i]
            self.maxHeapify(largest)

    def buildMaxHeap(self):
        for i in range(self.heapSize//2 - 1, -1, -1):
            self.maxHeapify(i)

    def heapsort(self):
        self.buildMaxHeap()
        for i in range(self.heapSize - 1, 0, -1):
            self.a[0], self.a[i] = self.a[i], self.a[0]
            self.heapSize -= 1
            self.maxHeapify(0)
        self.heapSize = len(self.a)
        return self.a
```

Input/Output: HackerRank Format

input(), map(int, array), etc.

```
# in Python 2, raw_input() is used; it returns a string
s = input()
n = int(input())

int_list = map(int, input().split())
```

Iteration: accumulate builds up partial results

itertools.accumulate produces accumulated sums. If a function of two arguments is given, it is applied to the first and second items, then to this result and the third item, and so on.

```
a = [1, 2, 3, 4, 0, 6]
list(accumulate(a, lambda a, b: a * b))
# [1, 2, 6, 24, 0, 0]
```

Iteration: chain joins given iterables

itertools.chain joins its arguments in order. chain.from_iterable(iter) joins the iterables within the given iterable.

```
list(chain('ABC', range(1, 4)))
# ['A', 'B', 'C', 1, 2, 3]

list(chain(enumerate("ABC")))
# [(0, 'A'), (1, 'B'), (2, 'C')]
```

Iteration: compress keeps values for corresponding True values

itertools.compress consumes two iterables in parallel, returning the values of the first argument for which corresponding values of the second argument are True. The returned object is an iterator. 1 and 0 may be used instead of True and False

```
a = compress([1,2,3,4,5], [True, False, False, True, True])
list(a)  # [1, 4, 5]
```

Iteration: Coroutines

Syntactically, coroutines look like generators. However, in a coroutine, yield will usually appear on the right side of an assignment. Unlike generators, you can both send and receive data to a coroutine.

Iteration: count produces numbers by steps

itertools.count returns a generator. Does not end.

```
gen = count(1, 0.5)
next(gen)  # 1
next(gen)  # 1.5
next(gen)  # 2.0
```

Iteration: cycle repeatedly

itertools.cycle(iter) saves a copy of each item in iter and repeatedly produces them without end.

```
list(islice(cycle(range(1, 4)), 10))
# [1, 2, 3, 1, 2, 3, 1, 2, 3, 1]
```

Iteration: dropwhile predicate is True

itertools.dropwhile evaluates the predicate for items in the second argument. Once it is False, the remaining items are returned, and no more items are checked by the predicate.

```
d = dropwhile(lambda n: n < 3, count(1, 0.5))
next(d)  # 3.0
next(d)  # 3.5
```

Iteration: enumerate() pairs an index with the corresponding element

enumerate(iter, start=0)

```
squares = [0, 1, 4, 9, 16, 25, 36, 49]

for (i, sq) in enumerate(squares):
    if sq % 2 == 1:  # if square is odd
        print("The square of", i, "is odd")
```

Iteration: Fibonacci sequence

```
def fib_iter(n):
    """Iterative version of Fibonacci sequence"""
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

def test():
    """[0, 1, 1, 2, 3, 5, 8]"""
    testeql(fib_iter(0), 0)
    testeql(fib_iter(1), 1)
    testeql(fib_iter(3), 2)
    testeql(fib_iter(6), 8)
```

Iteration: Generator as __iter__

Any function that has `yield` is a generator. yield may be used more than once in a generator definition.

```
import re

class Sentence:
    def __init__(self, text):
        self.text = text
        self.words = re.findall(r'\w+', text)

    def __iter__(self):
        for word in self.words:
            yield word
```

Iteration: genexps (generator expressions)

```
import re

class Sentence:
    def __init__(self, text):
        self.text = text
    def __iter__(self):
        return (match.group() for match in re.finditer(r'\w+', self.text))

s = Sentence("a man a plan a canal panama")
i = iter(s)
list(s)
```

Iteration: Group sequence into subsequences

```
size = 3
a = 'abcdefghijklmnopqrstuvwxyz'
[a[size*i:size*(i+1)] for i in range(len(a)//size+1)]
# ['abc', 'def', 'ghi', 'jkl', 'mno', 'pqr', 'stu', 'vwx', 'yz']
```

Iteration: groupby a given key

itertools.groupby(iter, key=None) produces tuples in the form (key, group) where key is the grouping criterion and group is the generator producing the corresponding items. Grouped items must be placed together in given iter.

```
animals = ['bee', 'cat', 'duck', 'dog', 'tiger', 'sheep']
animals = sorted(animals, key=len)
[(length, list(group)) for (length, group) in groupby(animals, len)]
# [(3, ['bee', 'cat', 'dog']), (4, ['duck']), (5, ['tiger', 'sheep'])]
```

Iteration: Interleave two lists

Note: lists must have the same length.

```
from itertools import chain

a = [1, 2, 3]
b = ['a', 'b', 'c']

c = chain(*zip(a, b))
list(c)  # [1, 'a', 2, 'b', 3, 'c']
```

Iteration: islice for iterables

itertools.islice(it, stop) and islice(it, start, stop, step=1) works for any iterable and is lazy

Iteration: Iterable, definition

An iterable is any object from which the built-in function iter can obtain an iterator. Sequences are always iterable. While iterators are iterable, iterables are not iterators. Do not define __next__ and __iter__ in the same class, making it an iterable and iterator at the same time. One should be able to instantiate multiple, independent iterators from the same object.

```
class C:
    def __init__(self):
        self.items = ['a','b']
    def __getitem__(self, index):
        return self.items[index]

c = C()
i = iter(c)
next(i)  # 'a'
next(i)  # 'b'
next(i)  # StopIteration
```

Iteration: iterate until a sentinel is reached

iter(callable, sentinel) returns an iterator that stops when the sentinel was supposed to be returned.

```
from random import randint

def d6():
    return randint(1, 6)
```

```
list(iter(d6, 1))
# [6, 3, 4, 2, 3, 6] : 1 is not present
```

Iteration: Iterator, definition

An iterator is any object that implements __next__, returning the next item in the sequence, and raises StopIteration when there are no more items. Iterators also implement __iter__, making them iterable. An iterator cannot be rewound; a new one (with initial state) must be created. The __iter__ method in an iterator may be: return self

Iteration: repeat an item

repeat(item, times=[forever]) repeats the given item the number of times given

```
list(repeat(9, 3))
```

Iteration: reversed

reversed(seq) returns a reverseiterator of a sequence or object that implements __reversed__

```
reversed([1,2,3])
```

Iteration: takewhile predicate is True

An itertools generator that consumes another generator and stops when the given predicate is False

```
gen = takewhile(lambda n: n < 3, count(1, 0.5))
list(gen)  # [1, 1.5, 2.0, 2.5]
```

Iteration: tee returns n generators

itertools.tee(iter, n=2) returns n independent generators that produce the items in iter.

```
g1, g2 = tee('ABC')
```

Iteration: Why sequences are iterable

To iterate over an object x, `iter(x)` is called. This call checks if `__iter__` is implemented. If not, __getitem__ is called, starting with the index 0. If neither methods are implemented, TypeError is raised.

Iteration: yield from

yield from allows a generator to delegate part of its operations to another generator. For simple iterators, it replaces a for loop

```
# for simple generators, yield from iterable is equivalent to:

for item in iterable:
    yield item
```

Linked Lists: List Node

```
class ListNode(object):
    def __init__(self, x):
        self.value = x
        self.next = None
```

Lists: 1D Initialization

```
size = 5

arr = [0] * size

# Do not use this pattern with reference values,
# such as arr = [another_arr] * 9
# Changes to one part will also occur in any other
# corresponding places.
```

Lists: 2D Initialization

```
# arr_2d[row][col]
rows = 3
cols = 5

arr_2d = [[0 for _ in range(cols)] for _ in range(rows)]
```

Lists: 3D Initialization

```
# arr_3d[layer][row][col]
layers = 2
rows = 3
cols = 4

arr_3d = [[[0 for _ in range(cols)] for _ in range(rows)] for _ in range(layers)]
```

Lists: Deep copy

```
import copy

a = [2, 3]
b = [1, a, (4, 5)]
c = copy.deepcopy(b)
```

Lists: Shallow copy

A shallow copy duplicates the references found in the outermost collection.

```
a = [3, [5, 4], (7, 8, 9)]
b = list(a)  # shallow copy
c = a[:]  # also a shallow copy

import copy
d = copy.copy(a)  # another way of making a shallow copy
```

Lists: Slicing

```
a = [1, 2, 3]
r = list(range(2, 11))  # ranges do not include endpoint; ends at 10

# r[20] raises IndexError
r[::-1]    # reverse
r[1:20:2]  # step of 2, out of bounds endpoint does not raise Error
```

Lists: Transpose a 2D list

Use list() on zip(), if needed

```
# if the list is perfectly rectangular
arr_tr = zip(*arr)

# otherwise, to fill in spaces
```

```
from itertools import zip_longest

arr_tr = zip_longest(*arr, fillvalue=0)  # or ' ', etc.
```

Lists: Zip two lists together

Zip ends with shortest list, use zip_longest to fill in blanks

```
names = ['Joe', 'Alice', 'Ken', 'Tim', 'Sarah']
scores = [68, 72, 99, 74, 75]

# a student passes with a score of 75 or higher
[(name, score >= 75) for (name, score) in zip(names, scores)]

from itertools import zip_longest

trees = ['elm', 'ash', 'fir']
heights = [78, 62]

print(list(zip_longest(trees, heights, fillvalue=0)))
```

Loops: While loops: know where variables are when they end

```
n = 0
while n < 10:
    n += 1

print(n)  # 10
```

Loops: While loops: use while True as alternative to do-while

There is no do-while loop in Python. To avoid writing update conditions twice, put a condition inside the while loop to break out.

Mathematics: Exponentials and logarithms

```
(a ** m) ** n == a ** (m * n)
(a ** m) * (a ** n) == a ** (m + n)

In Python, log(n) is the natural logarithm

a == b ** (log_b(a))
log(a * b) == log(a) + log(b)
log(a ** n) == n * log(a)
log_b(a) == log_c(a) / log_c(b)  # change of base, c can be any base
log(1 / a) == -log(a)
log_b(a) == 1 / log_a(b)
a ** (log_b(c)) == c ** (log_b(a))
```

Matrices: Transpose a matrix

```
a = [[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
tr = zip(*a)

list(tr)
# [(1,4,7,10), (2,5,8,11), (3,6,9,12)]
```

Metaprogramming: Descriptors

A descriptor is an object attribute with binding behavior, whose attribute access has been overridden by methods in the descriptor protocol. These methods are __get__, __set__, and __delete__

Metaprogramming: Dynamic attributes

The special methods __getattr__ and __setattr__ are called to evaluate access to attributes.

Number Manipulation: Change a decimal integer to arbitrary base

Valid for 2 <= base <= 9

```
def changeBase(n, base):
    digits = []

    while n > 0:
        digits.append(n % base)
        n //= base
    return ''.join(map(str, digits))[::-1]
```

Number Manipulation: Convert arbitrary base to decimal

```
n = '3eg'  # some number in base 19
d = int(n, 19)  # 1365 in decimal

# If coding literally, write:
h = 0xfe
o = 0o755
b = 0b1101
```

Number Manipulation: Convert decimal to bin, oct, hex

```
n = 123
b = bin(n)[2:]  # discard initial 0b
o = oct(n)[2:]
h = hex(n)[2:]
```

Number Manipulation: Convert integer to list of digits

```
n = 12345
digits = list(map(int, str(n)))
```

Number Manipulation: Convert list of ints to string

```
a = [1, 2, 3]
''.join(map(str, a))
```

Number Manipulation: Process integer's digits right-to-left

Given a decimal integer, do something digit-by-digit, starting from ones (right side)

```
n = 12345
while n > 0:
    digit = n % 10
    print(digit)
    n //= 10
```

Number Theory: Compositeness Test (pseudoprimes)

Check if a given number is probably prime. Carmichael numbers are composite, but will fool Fermat's test (it is a flawed test).

```
def isProbablePrime(n):
    if n == 2:
        return True
    if not n & 1:
```

```
        return False  # even numbers
    return pow(2, n-1, n) == 1
```

Number Theory: GCD (Euclid's algorithm)

```
from fractions import gcd

def sicp_gcd(a, b):
    if b == 0:
        return a
    return sicp_gcd(b, a % b)
```

Number Theory: LCM (using GCD)

```
from fractions import gcd

def lcm(a, b):
    return (a * b) // gcd(a, b)
```

Number Theory: Sieve of Eratosthenes

```
def primes(n):
    if n < 2:
        return []
    isPrime = [True] * (n+1)  # begin with all numbers prime
    for base in range(3, int(n ** 0.5) + 1, 2):  # potential primes
        # stop at sqrt(n) because the larger number would
        # have already been crossed out
        for multiple in range(base * 2, n+1, base):
            isPrime[multiple] = False
    primeList = [2]  # manually include the only even number
    for n in range(3, n+1, 2):  # consider all odd numbers
        if isPrime[n]:
            primeList.append(n)
    return primeList

def test():
    testeql(primes(73), [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73])
```

Object-Oriented Programming: __slots__

__slots__ allows you to save memory. However, they are not inherited by subclasses. To define __slots__ is to say, "These are all the instance attributes for this class." Another downside of __slots__ is that an user will not be able to add other attributes to instances. To circumvent this, add '__dict__' to the __slots__ tuple. __weakref__ may also be added.

```
class Vector2d:
    __slots__ = ('__x', '__y')
```

Object-Oriented Programming: API, definition

An API (Application Programming Interface) is the set of functions, protocols, and tools for building software. It allows a programmer to use and access code written by the API's author.

Object-Oriented Programming: Attribute access

Implementing __getattr__ allows you to customize the result of `instance.x`

__getattr__ is only called when the attribute, x, does not exist. If x is assigned to the instance, __getattr__ will no longer be called. __setattr__ must be defined to take care of this scenario.

Object-Oriented Programming: classmethod (decorator)

@classmethod defines a method that operates on the class. The class itself (as `cls`) is received as the first argument, instead of what's typically the instance. This decorator is typically used for alternative constructors

Object-Oriented Programming: Extend a built-in class

```
class AthleteList(list):
    def __init__(self, a_name, a_dob=None, a_times=[]):
        # list.__init__([])  # appears to be optional
        self.name = a_name
        self.dob = a_dob
        self.extend(a_times)
    def top3(self):
        return sorted(set(self))[:3]
```

Object-Oriented Programming: Goose typing

Goose typing is using `isinstance(obj, cls)`, where cls is an abstract base class (ABC). In other words, cls' metaclass is abc.ABCMeta

Object-Oriented Programming: Hash value

Use XOR (^) to combine the hashes of the object's components.

```
# in class Vector2d, with properties x and y

def __hash__(self):
    return hash(self.x) ^ hash(self.y)
```

Object-Oriented Programming: Instances may be callable

Defining __call__ inside a class definition allows instances of that class to be called.

Object-Oriented Programming: Interface and protocol, definition

An interface is a set of method definitions. It is the subset of an object's public methods that allows it to play a specific role in the system.

A protocol is an informal interface. They are independent of inheritance. Protocols cannot be verified statically by the interpreter.

An analogy in the real world is the controls of a car. The interface of a car is the steering wheel, pedals, horn, and other controls.

Object-Oriented Programming: is compares identity and == value

`is` and `is not` compares the identity of objects, while == will compare their values

```
alex_a = { 'name': 'Alex', 'born': 1990 }
alex_b = { 'name': 'Alex', 'born': 1990 }

id(alex_a)  # 17510012
id(alex_b)  # 17510274

alex_a == alex_b  # True
alex_a is alex_b  # False

alex_a is not alex_b  # True
```

Object-Oriented Programming: MRO (Method Resolution Order)

The __mro__ attribute in a class is a tuple in the MRO order. To call a specific superclass method, call the class method and pass the instance as its first argument.

```
class B():
    def pong():
        print("PONG", self)

class C():
    def pong():
        print("PONG", self)

class D(B, C):
    def ping(self):
        print("PING", self)

d = D()
C.pong(d)
```

Object-Oriented Programming: Name mangling

Prefixing an attribute with two underscores will cause the attribute to become _ClassName__AttrName behind the scenes.

It may be better to explicitly use _ClassName_AttrName (single underscore prefix), because a single underscore has no special meaning. However, at a module level, names with a single underscore prefix will not be imported.

Object-Oriented Programming: Polymorphism, definition

A language feature that allows values of different types to be handled by a uniform interface.

Object-Oriented Programming: Protocols

A protocol is an informal interface. For example, the protocol of a sequence implies onlt __len__ and __getitem__. Duck typing is calling an object a sequence because it behaves like one, not specifically because it is a subclass of sequence.

Object-Oriented Programming: Read-only attributes

```
class Vector2d:
    def __init__(self, x, y):
        self.__x = float(x)  # use double underscore prefix
        self.__y = float(y)

    @property
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y
```

Object-Oriented Programming: staticmethod (decorator)

@staticmethod alters a method so that it doesn't receive the special first parameter. It is like a simple function that happens to be inside a class definition.

Object-Oriented Programming: Subclass UserCollections instead of built-ins

Subclassing built-ins directly is unreliable because overwritten methods are not called. Instead, subclass UserList, UserDict, and UserString from the collections module.

Object-Oriented Programming: Subclasses must explicitly call super's __init__

There is a difference between __init__ and __new__. The superclass' __init__ must be called explicitly.

Object-Oriented Programming: super()

`super()` allows you to access methods from superclasses.

Object-Oriented Programming: Use is to compare with None

Instead of using ==, use `is` and `is not` to check if a variable is or is not None

Object-Oriented Programming: Virtual subclass

A virtual subclass does not inherit from a superclass, but it is registered as `TheSuperClass.register(TheSubClass)`, or with a decorator `@TheSuperClass.register`

Object-Oriented Programming: Weakrefs

A weak reference does not increment the reference counter, so the object it refers to may be deleted when all its strong references are gone. Weakrefs are useful for caches. Weakref collections should be used instead of `weakref.ref` directly (WeakValueDictionary, WeakKeyDictionary, and WeakSet)

lists, dicts, ints, and tuples cannot be the target of a weakref, but user-defined types can. While a subclass of list may be the target of a weakref, a subclass of int or tuple cannot.

```
import weakref

class Cheese:
    def __init__(self, kind):
        self.kind = kind

stock = weakref.WeakValueDictionary()
catalog = [Cheese('Tilsit'), Cheese('Brie'), Cheese('Parmesan')]

for cheese in catalog:
    stock[cheese.kind] = cheese

del catalog
sorted(stock.keys())  # ['Parmesan']

del cheese  # the temporary variable is a strong reference
sorted(stock.keys())  # []
```

Python Syntax: Assign by capturing a group of items

```
a, b, *rest = range(5)
# 0, 1, [2, 3, 4]

a, *middle, end = range(5)
# 0, [1, 2, 3], 4
```

Python Syntax: Context managers

The contextlib module contains several utilities. Some of them are: closing (for objects that implement close()), suppress (ignore specific exceptions), @contextmanager (create a context manager from a simple generator), ContextDecorator (a base class for context managers), and ExitStack (exit multiple context managers).

Python Syntax: Data Structure Literals

```
a_list = [1, 2, 3]
a_set = {1, 2, 2, 3}  # set() is an empty set
a_dict = {'a': 1, 'b': 2}  # {} is an empty dict

type(a_list)  # get type
```

Python Syntax: else with control structures

An else block after a for or while loop will be executed if the loop exits normally (that is, was not interrupted by a break). After a try block, else will be executed if no exception was raised. A better keyword would be 'then'.

Python Syntax: Scope of variables, global

```
b = 6

def f(a):
    global b  # refer to global variable outside this function
    print(a)
    print(b)
    b = 9

f(3)  # 3 ; 6
b     # 9
```

Python Syntax: Scope of variables, nonlocal

nonlocal allows you to assign to variables in an outer, but not global, scope

Python Syntax: Unpacking a list

```
t = (20, 8)
divmod(*t)

from itertools import product

strs = ['abc', 'def', 'ghi']
list(product(*strs))
```

Python Syntax: Variable typing

Python is strongly typed and dynamically typed. In a weakly typed language, variables may be implicitly converted to a diffent type (PHP, JavaScript). In a statically typed language, type checking is done at compile time.

Queues: deque initialization

```
from collections import deque

d = deque([3, 5, 2])
d.append(9)
left = d.popleft()  # 3
d.appendleft(1)
# deque([1, 5, 2, 9])

# maxlen is optional
d_fixed_size = deque(range(20), maxlen=5)
# deque([15, 16, 17, 18, 19], maxlen=5)

d_fixed_size.appendleft(0)
# deque([0, 15, 16, 17, 18], maxlen=5)
```

Random Numbers: Floating-point, generating a random value

```
from random import uniform

n = uniform(0.5, 2.5)  # the endpoint may not be included
```

Random Numbers: Integer, generating a random value

```
from random import randint

n = randint(1, 10)  # both endpoints are included
```

Random Numbers: Pick a random element

```
from random import choice

a = [1, 2, 3, 4, 5, 6]
choice(a)
```

Random Numbers: Shuffle an array

```
from random import shuffle

a = list(range(10))
shuffle(a)
a  # [7, 5, 3, 8, 0, 4, 2, 9, 6, 1]
```

Random Numbers: SystemRandom

A class that creates random bytes suitable for use in cryptography (if the underlying OS supports it)

```
from random import SystemRandom

sr = SystemRandom()
n = sr.randint(1, 10)
```

Regex: Backreferences

A backreference allows you to specify that the contents of an earlier capturing group must also be found at the current location. \1 will succeed if the contents of group 1 can be found at the current position.

```
p = re.compile(r'(\b\w+)\s+\1')
p.search("Paris in the the spring").group()  # 'the the'
```

Regex: Compiling or using module-level functions

Compiling a pattern will save some function calls. Flags control the behavior of the RE.

ASCII (A) \w, \b, \s, \d match only ASCII characters

DOTALL (S) . matches any character, including newlines

IGNORECASE (I) case-insensitive matches

LOCALE (L) locale-aware match

MULTILINE (M) multi-line matching, affecting ^ and $

VERBOSE (X) enable verbose REs, which may be organized more clearly

```
import re

pat = re.compile(r'Froms+')
pat.match('From amk')  # <_sre.SRE_Match object ...>

re.match(r'Froms+', 'From amk')  # <_sre.SRE_Match object ...>
```

Regex: Groups (parenthesized)

A match can be divided into groups, indicated by parentheses. Groups are numbered by counting opening parentheses from left to right.

matchObject.groups() returns a tuple of the strings from group 1 and up.

```
p = re.compile('(a(b)c)d')
m = p.match("abcd")
m.group(0)  # 'abcd'
m.group(1)  # 'abc'
m.group(2)  # 'b'
```

Regex: Lookahead assertions

(?=...) is a positive lookahead assertion. It succeeds if the contained RE ... successfully matches at the current location. The matching engine does not advance. The rest of the pattern is tried where the assertion started.

(?!...) is a negative lookahead assertion. It succeeds if the RE ... doesn't match at the current position.

Regex: Match object

When a match is found, a match object is returned.

```
pat = re.compile(r'[a-z]*')
m = pat.match("joe123")
m.group()  # "joe"
m.start(), m.end()  # (0, 3)
m.span()  # (0, 3)
```

Regex: Matching

match() determines if the RE matches at the beginning of the string

search() looks for a match anywhere in the string

findall() finds all substrings and returns them as a list

finditer() finds all substrings and returns them as an iterator

Regex: Named groups

(?P<name>...) allows you to retrieve the group's contents by its name

```
p = re.compile(r'(?P<word>\b\w+\b)')
m = p.search('(( lots of punctuation ))')
m.group('word')  # 'lots'
```

Regex: Non-capturing group

(?:...) where ... is a regex, denotes a part of a RE, but where we are not interested in the group's contents.

Regex: Raw string

A raw string is prefixed with an r. Everything is taken literally, so there is no need to escape backslashes.

```
s = r'\\section'
```

Regex: Repetition

* represents greedy repetition of the preceding character (zero or more times)

+ matches one or more times

? matches either zero or one time

{m,n} matches at least m and at most n times (endpoints are included)

To make a qualifier non-greedy, add a ? to its right, *?, +?, ??, and {m,n}?. They will match as little text as possible.

Regex: Special sequences

The capitalized version of a special sequence is the inverse (\d are digits, \D are non-digits).

`\d` is any decimal digit, [0-9]

`\s` is any whitespace character, [\t\n\r\f\v]

`\w` is any alphanumeric character, [a-zA-Z0-9_]

| 'or' operator

^ matches the beginning of lines

$ matches the end of a line

\A matches the start of the string

\Z matches the end of the string

\b word boundary

Regex: Splitting a string

split(string, maxsplit=0) will split the string where the RE was found.

```
p = re.compile(r'd+')
p.split('a1b2c3')  # ['a', 'b', 'c', '']
```

Regex: Substitutions

Replace all matches with a given replacement value

sub(replacement, originalString, count=0)

subn(replacement, originalString, count=0) does the same thing, but returns a tuple of the new string and the number of replacements

```
p = re.compile(r'(blue|white|red)')
p.sub('color', 'blue socks and red shoes')
```

```
# 'color socks and color shoes'
```

Searching: Binary search

```
def binarySearch(arr, x):
    # returns the index of x in arr, or -1 if not found
    # arr must be sorted
    left = 0
    right = len(arr) - 1
    while True:
        if right < left:
            return -1

        midpoint = (left + right) // 2
        if arr[midpoint] < x:
            left = midpoint + 1
        elif arr[midpoint] > x:
            right = midpoint - 1
        else:
            return midpoint
```

Sets: Apply functions to many iterables

The union of four sets, a, b, c, and d, can be computed with one function call

```
a.union(b, c, d)
```

Sets: Elementwise modifications

```
s.add(e)      # add e to s
s.clear()     # remove all elements from s
s.copy()      # returns a shallow copy of s
s.pop()       # removes and returns an arbitrary element
s.discard(e)  # removes e if it exists, otherwise does nothing
s.remove(e)   # removes e if it exists, otherwise throws KeyError
s.update(t)   # update s with the union of itself and t. t may be a list
```

Sets: frozenset can be included in other sets

Because a set can only include hashable items, a set cannot be included in another set. However, a frozenset is hashable.

Sets: Operations

```
s = {1, 2}
t = {1, 2, 3, 4, 5}
u = {1, 2, 3, 4, 5}

n = 1

n in s      # True
n not in s  # False

s < t   # Is s a proper subset of t? True
u < t   # False, because r == t

s <= t  # Is s a subset of t? True

s > t   # Is s a proper superset of t? False
s >= t  # Is s a superset of t? False

== # Equals
|  # Union
&  # Intersection
-  # Difference
```

```
^  # symmetric difference, elements in either set
   # but not in both
```

Sets: Set literal

```
s = {1, 2, 3}
t = {1}
empty_is_a_dict = {}
empty_set = set()
```

Sets: setcomps (set comprehensions)

```
s = {x for x in range(10) if x % 2 == 1}  # odd numbers
```

Sorting: Bubble Sort

```
def bubbleSort(arr):
    for i in range(len(arr) - 1):
        for j in range(len(arr) - 1, i, -1):
            if arr[j] < arr[j-1]:
                arr[j], arr[j-1] = arr[j-1], arr[j]
    return arr
```

Sorting: Insertion Sort

The numbers we wish to sort are the keys. The procedure is like sorting a hand of playing cards. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left. Insertion sort rearranges the array in place.

```
def insertionSort(arr):
    # begin at the second element
    for j in range(1, len(arr)):
        key = arr[j]
        # insert arr[j] into the sorted sequence arr[0:j]
        i = j - 1
        while i >= 0 and arr[i] > key:
            arr[i+1] = arr[i]
            i -= 1
        arr[i+1] = key
    return arr
```

Sorting: Merge Sort

```
def mergeSort(arr):
    n = len(arr)
    if n > 1:
        middle = n // 2
        left = arr[:middle]
        right = arr[middle:]

        mergeSort(left)
        mergeSort(right)

        # merge
        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1

        while i < len(left):
```

```
            arr[k] = left[i]
            i += 1
            k += 1

        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1
    return arr
```

Sorting: Quicksort

```
def quicksort(arr):
    def partition(left, right):
        i = left - 1
        for j in range(left, right):
            if arr[j] <= arr[right]:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
        arr[i+1], arr[right] = arr[right], arr[i+1]
        return i + 1

    def quicksortHelper(left, right):
        if left < right:
            middle = partition(left, right)
            quicksortHelper(left, middle - 1)
            quicksortHelper(middle + 1, right)

    quicksortHelper(0, len(arr) - 1)
    return arr
```

Sorting: Radix Sort

Use a stable sorting algorithm to sort by the ones place, then the tens place, and so on.

```
from operator import itemgetter

def radix_sort(lst):
    # set up the list, separating numbers into digits
    lst_str = list(map(str, lst))
    len_longest = len(max(lst_str, key=len))
    lst_pad = [s.zfill(len_longest) for s in lst_str]

    for i in range(len_longest - 1, -1, -1):
        lst_pad.sort(key=itemgetter(i))
    return list(map(int, lst_pad))
```

Sorting: Sort a mixed list of strings and integers

```
a = [28, 14, '28', '23', 20]
s = sorted(a, key=int)  # [14, 20, '23', 28, '28']

# downside: must then convert resulting list to a uniform type
```

Sorting: Sort list of lists by column

```
from operator import itemgetter

w = [[12, 'tall', 'blue', 1],
     [2, 'short', 'red', 9],
     [4, 'tall', 'blue', 13]]
s = sorted(w, key=itemgetter(1, 0))
```

Sorting: Sort namedtuples by attributes

```
from collections import namedtuple
from operator import attrgetter

Person = namedtuple('Person', 'name age')
people = [Person("Joe", 34), Person("Amy", 34),
          Person("Jen", 52)]
ps = sorted(people, key=attrgetter('age', 'name'))
```

Sorting: sorted() function

```
optional parameters:
reverse=True
key=str.lower  # function of one argument applied to each item
```

SQL: Inner Join

Retrieve an employee's name and heis or her department

```
SELECT e.fname, e.lname, d.name
FROM employee AS e INNER JOIN department AS d
ON e.dept_id = d.dept_id
```

Stacks: Lists behave like LIFO stacks

Add to a stack with `append()` and remove with `pop()`

```
stack = [3, 4]
stack.append(5)  # [3, 4, 5]
stack.pop()  # 5
```

String Formatting: %-formatting

Using % as placeholders is called the "old %-Formatting"

```
%s string
%d integer
%f float

name = "World"
print("Hello, %s" % name)
```

String Formatting: format() function

```
'{1} {0}'.format('one', 'two')  # 'two one'
```

Strings: Case-insensitive comparisons

When dealing with non-ASCII characters, lower() is not reliable

```
s = "AbCdE"
s.casefold()  # 'abcde'
```

Strings: Check if upper, lower, digit, etc.

Given a string, check if all its characters are uppercase, lowercase, digits, etc.

```
s = "my string"
s.islower()
s.isupper()
s.isdigit()
```

Strings: Convert to upper and lowercase

```
s = "PyThOn iZ kEwL"
s.upper()  # PYTHON IZ KEWL
s.lower()  # python iz kewl
s.swapcase()  # pYtHoN Iz KeWl
```

Strings: Encode and decode (strings to bytes)

```
s = 'café'
b = s.encode('utf-8')
d = b.decode('utf-8')
```

Strings: Generate the alphabet

```
alphabet = ''.join([chr(i) for i in range(ord('a'), ord('z')+1)])
```

Strings: Remove diacritics

```
import unicodedata
import string

def remove_diacritics(s):
    norm_s = unicodedata.normalize('NFD', s)
    result = ''.join(c for c in norm_s if not unicodedata.combining(c))
    return unicodedata.normalize('NFC', result)
```

Strings: Reverse

```
"abc"[::-1]

s = "esrever"
r = s[::-1]
```

Strings: Unicode normalization

Normalization of Unicode strings allow for safer comparisons between them

```
from unicodedata import normalize

s = "café"
normalize('NFC', s)  # 'café'

# NFC combines characters as much as possible
# (resulting in the shortest string)

# NFD decomposes characters into basic ones (letters and diacritics)

# NFC is recommended by the W3C

# NFKC and NFKD (the K stands for Compatibility) are stronger forms
# of normalization

# NFKC and NFKD cause loss of data (4^2 becomes 42) so must be used
# with caution
```

Testing: assert (simple testing)

```
def f(x):
    return x + 1

def test():
    """
    Message string after assertion is optional, will appear on failure.
    Display "OK" at the end to indicate success.
    Call test() in interactive session.
```

```
    """
    assert f(99) == 100, "f of 99"
    assert f(f(9)) == 11, "Chain of f"
    assert f(f(0)) == 2
    print("OK")
```

Trees: BFS (Breadth-First Search)

See Graphs: BFS

Trees: Binary Tree traversal

A binary tree consists of a record with a left and right branch, and may be traversed recursively. Starting on the root node, draw a curve counterclockwise (moving to the left at first). In preorder traversal, we print the node as the curve touches its left edge. For postorder, we use each node's right edge. For inorder, we use each node's bottom edge.

```
class Tree:  # a single node is a tree
    def __init__(self, value, left, right):
        self.value = value
        self.left = left
        self.right = right

    def preorder(self):
        print(self.value, end=" ")
        if self.left:
            self.left.preorder()
        if self.right:
            self.right.preorder()

    def postorder(self):
        if self.left:
            self.left.postorder()
        if self.right:
            self.right.postorder()
        print(self.value, end=" ")

    def inorder(self):
        if self.left:
            self.left.inorder()

        print(self.value, end=" ")

        if self.right:
            self.right.inorder()
```

Trees: DFS (Depth-First Search)

See Graphs: DFS

Trees: Red-Black Tree

A red-black tree is a self-balancing binary search tree. The color property of each node is used to balance the tree upon insertions or deletions. Rebalancing involves recoloring and rotating the tree.

Tries: Trie for words

A trie is a tree with an empty node at its root (typically). For representing words, each child node holds a letter. To identify a word as a 'search hit', we assign a value such as 5 for the node holding the last letter of the word.

Tuples: ids of elements never change

Although tuples are considered immutable, in reality only the ids of the elements they contain never change. A tuple may contain lists, and these may be changed in-place.

```
t1 = (1, 2, [30, 40])
t2 = (1, 2, [30, 40])

t1 == t2  # True

t1[-1].append(99)
t1 == t2  # False
```

Tuples: namedtuple

A lightweight, dictionary-like object. Fields are accessed with dot notation.

```
from collections import namedtuple

Card = namedtuple("CardDisplay", 'rank suit')
# alternatively namedtuple("CardDisplay", ['rank', 'suit'])

big_two = Card(2, 'Spades')
print(big_two.suit)

print(big_two._asdict())
```

Tuples: Shortcuts for copying will return the same tuple

Two shortcuts for creating shallow copies of lists will not work for tuples. They will return the reference to the same tuple

```
t = (1, 2)
t2 = tuple(t)  # t2 is t will be True
t3 = t[:]  # t3 is t will be True
```