BUILT-IN FUNCTIONS & MISC		
Purpose	**Example**	
Ceiling	`math.ceil(1.5) = 2`	Round up its argument
Floor (//)		Round down to integer (Quotient)
Modulo (%)		(Remainder)
Backslash (\)	`print('That's') XXX` `print('That\'s')`	Escapes normally special characters
Import	`from math import *` `import math`	(good if redefining previously defined functions e.g. max)
Round off to 3 d.p.	`round(ans,3)`	
Range	`range(start, stop, step)`	By default: `start` : 0 `step` : 1

Iteration	`continue`
	
	`break`
		Continues the next iteration of the loop/ next value
		Break out of the loop

Lambda	`(lambda x: x+1)(5) = 6`
	`lambda x: int(x)`
	`lambda: x`
		The lambda function is like f.
		Same as `int`, a function too
		Mask/wrap value in function

List comprehension	`_ (for _ in _) (if _ else _)`	
Random (`from random import *`)	`random()`	Interval [0,1] E.g. `0.6119987281172101`
	`random.uniform(low, high)` `random.randint(a, b)`	Inclusive ends

gcd	`from fractions import gcd`
	
	`g = gcd(n, d)`	

Reading a file	`input =`
	`open('inputfilename.txt', 'r')`
	
	`some_line = input.readline()`
		Opens up file for reading & reads line by line.
		End of file is empty string.

Writing to a file	`output =`
	`open('inputfilename.txt', 'w')`
	
	`output.write('HELLO WORLD')`	

Reading csv file (`import csv`)
- Returns a list of lists
- Each list is each row
	`def read_csv(csvfilename):`
	`  rows = []`
	`  with open(csvfilename) as csvfile:`
	`  file_reader = csv.reader(csvfile)`
	`  for row in file_reader:`
	`  rows += (list(row),)`
	`  return rows`	

Operators	`import operator`
	`ops = {"+": operator.add,`
	`"-": operator.sub,`
	`"*": operator.mul,`
	`"/": operator.truediv}`
		Note: `operator.truediv` gives a float (not int)

Nonlocal	`nonlocal value`	Refers to outerscope value

* notation	`def sum_all(*args):`
	`  result = 0`
	`  if args is not None:`
	`  for arg in args:`
	`  result += arg`
	`  return result`
		In a tuple

STRING / TUPLE		
String slicing	`s[start:stop:step]`	`stop`: not inclusive `step = 2`: to skip an element `step` = (-)ve: steps taken in index
Tuple index slice		Valid range for tuple: -n <= x <= n - len(tuple) = n - (- n) for negative index slice
Obtain start position of string	`my_string.find('dog')` `S1.find(S2, position)`	Find string S2 in S1 starting from index position
	`seq.index(num)`	Only gives the index of first num it finds in the string/tuple
Replace string	`my_string.replace('dog','cat')`	Replace all occurrences of 'dog' in my_string to 'cat'
Reverse tuple	`tuple(reversed(tuple))` `tup[-1::-1]`	

Finding max or min	`max(a, b)`
	`max(a, b, c…)`
	`min(a, b)`	

	`element in x`	Returns True if element in x, False otherwise
	`for i in x`	
More than, less than	`str_1 > str_2` `tpl_1 < tpl_2`	Base on: 1) Alphabetical **(** 'b' > 'a')** 2) # of letters/element
Split	`tuple(line.split(','))`	Splits string of chara into tuple
`import string`	`string.ascii_uppercase` U65-90 `string.ascii_lowercase` U97-122	"ABCDEFGHIJKLMNOPQRSTUVWXYZ" "abcdefghijklmnopqrstuvwxyz" 97-122
	`ord()` `chr()`	Letter -> # (Unicode) # -> Letter
	`str.lower()`	Converts alphabets to lowercase
LIST		
Generates new list	`lst = [1, 2, 3, 4, 5]` **[** `n**2 for ele in lst` **]**	`a = "abc"` `list(a) == ["a", "b", "c"]`
Mutation Operations for list	**`Returns None`** `(Modifies same list)`	
	`lst.append(x)`	Add **element** x
	`lst.extend(x)`	Add another list x (a += b)
	`lst.reverse()`	Reverses the list
	`lst.insert(i, x)`	Insert element x at index i
	`lst.remove(x)`	Removes first occurrence of element x
	`lst.clear()`	Empties the list
	`del lst[i]`	Delete element at specific index
	`Returns a value`	
	`lst.copy()` `b = a[:]`	Returns shallow copy of the list `a[:]` equivalent to `a.copy()`
	`lst.pop()`	Removes last element of list and returns it
	`lst.pop(i)`	Removes element at index i of list and returns it
	`a = a + b`	Produces new list
Sorting	`sorted(a)`	- `a` can be a tuple or a list Creates a new sorted list

	`a.sort()`
	`a.sort(key = lambda x: x[1], reverse =True)`
	`a.sort(key = lambda x: (x[1], -x[2]))`
		Modifies list itself
		- Use `key` if don't want to sort in a natural way
		- `reverse = False`: sorts from smallest to largest

	`lst.count(score)`	

DICT		
Creating a dictionary	`dict([("a",1), ("b",2)])` `dict() == {}`	Needs to take in a sequence of pairs
Assignment	`<dict>[key] = value`	- If `key` exists, updates existing record - If none exists, creates new record

Deletion	`del <dict>[key]`
	`<dict>.clear()`
	`del <dict>`
	`del <dict>[key]`
		Deletes record corr. to key if exists
		Removes all entries in <dict>
		Deletes the dictionary <dict>
		Deletes key in dict

Access	`<dict>.get(key, default=None)` `<dict>.get(key, default="No")`	- If key in dict, return value. - Else, return default value.

	`key in <dict>`
	`<dict>.keys()`
	`<dict>.values()`
	`<dict>.items()`
	`len(<dict>)`
		Returns True if key in <dict>, else False
		Returns list of keys
		Returns list of values
		Returns list of (key, value) tuple pairs
		Returns number of elements in <dict>

CLASS		
	`isinstance(Truck(),Vehicle)` `type(Truck) == Vehicle`	

	`def say(self,stuff):`
	`  super().say(stuff +`
	`  self.fav_phrase)`
		Using method say from superclass but also redefining/modifying it.
		(If don't want then no need to put, just call it in super().__init__...)

Multiple inheritance	`class C(A,B):` `  def __init__(self,…`	Primarily class A, then class B
EXCEPTIONS		
	`SyntaxError`	

Exceptions (Errors detected during execution)	`Exception`
	`ZeroDivisionError`
	`NameError`
	`TypeError`
	`ValueError`
	`IndexError`
	`KeyError`
		Base of all errors
		
		Variable not defined
		E.g. "string" + 1
		E.g. int("one")
		
		Dictionary

	`try:`
	statements
	`return …`
	`except <ErrorType1>:`
	statements
	`except (<ErrorType2>,`
	`<ErrorType3>):`
	statements
	`except <ErrorType4> as err:`
	statements
	`raise MyError("Problem")`
	`except:`
	statements
	`else:`
	statments
	`finally:`
	`statements`
		If got error in "`try`" block, skip everything below and jump to "`except`".
		
		`err`: Error object that occurred
		
		`finally`: Always executed

	`def MyError(Exception):`
	`  def __init__(self,value):`
	`  self.value = value`
	`  def __str__(self):`
	`  return repr(self.value)`
		`try:`
		`raise MyError(2*2)`
		`except MyError as e:`
		`print("Exception value",`
		`e.value)`
		`>>> Exception Value: 4`

COMPLICATED ALGORITHMS	

Print out each character one at a time	`index = 0`
	`while index < len(fruit):`
	`  letter = fruit[index]`
	`  print(letter)`
	`  index += 1`

Taking last digit in a number	`num%10`

Finding prime num	`from math import *`
	
	`def is_divisible(x, i):`
	
	`def is_prime(x):`
	`  if x == 1:`
	`  return False`
	`  else:`
	`  for i in range(2, ceil(sqrt(x))):`
	`  if is_divisible(x, i):`
	`  return False`
	`  return True`

Count the # of occurrences (letter)	`def count_letter(string):`
	`  counter, index = 0, 0`
	`  for letter in string:`
	`  if letter == 'A':`
	`  counter += 1`
	`  return counter`

Count the # of occurrences (substring)
 	
- Counts substrings that begin with 'A' and end with 'X'
-E.g. CAXAAYXZA
- index = # of As
- counter: if X, add all the # of As before it
	`def count_substring(string):`
	`  counter, index = 0, 0`
	`  for letter in string:`
	`  if letter == 'A':`
	`  index += 1`
	`  if letter == 'X':`
	`  counter += index`
	`  return counter`

Count the number of occurrences of S2 in S1 (no overlap)
 	
E.g. 'CS1010S' has 2 occurrences of '10'
E.g. '110101' has only 1 occurrence of '101'
	`def occurrence(S1, S2):`
	`  counter, position = 0, 0`
	`  length = len(S2)`
	`  while position < len(S1):`
	`  position = S1.find(S2, position)`
	`  if position == -1:`
	`  break`
	`  counter += 1`
	`  return counter`

Fast Exponential
(b^e)
 	
Time: O(log n)
Space: O(log n)
	`def fast_expt(b,e):`
	`  if e == 0:`
	`  return 1`
	`  elif e%2 == 0:`
	`  return fast_expt(b*b, e/2)`
	`  else:`
	`  return b * fast_expt(b, e-1)`

TOWER OF HANOI
 	
1. Move n-1 discs from A to B using C
2. Move disc from A to C
3. Move n-1 discs from B to C using A
	`def move_tower(size,src,dest,aux):`
	`  if size == 1:`
	`  print_move(src, dest)`
	`  #display the move`
	`  else:`
	`  move_tower(size-1,src,aux,dest)`
	`  print_move(src, dest)`
	`  move_tower(size-1, aux, dest,src)`
	
	`def print_move(src, dest):`
	`  print("move top disk from", src,`
	`  "to", dest)`

	def hanoi(n, src, dsc, aux):
	  if n == 0:
	  return ()
	  elif n == 1:
	  return ((src,dsc) ,)
	  else:
	  return hanoi(n-1, src, aux, dsc)+
	  ((src,dsc),)+
	  hanoi(n-1,aux,dsc,src)

COUNT CHANGE
 	
1: 1 cent
2: 5 cent
3: 10 cent
4: 20 cent
5: 50 cent
6: 100 cent
 	
Time: leaves in tree
: $O(2^{a+d})$
: $O(2^{a})$
Space: depth of entire tree
: O(a)
 	
Where a = amount,
d = denomination (fixed)
	def cc(amount, kinds_of_coins):
	  if amount == 0:
	  return 1
	  elif (amount<0) or (kinds_of_coins<0):
	  return 0
	  else:
	  return
	  cc(amount -
	  first_denomination(kinds_of_coins)),
	  kinds_of_coins)+
	  cc(amount, kinds_of_coins -1)
	
	def first_denomination(kinds_of_coins):
	  if kinds_of_coins <= 0:
	  return None
	  elif kinds_of_coins == 1:
	  return 1
	  elif kinds_of_coins == 2:
	  return 5
	  ...

Counting leaves	def count_leaves(tree):
	  if tree == ():
	  return 0
	  elif is_leaf(tree):
	  return 1
	  else:
	  return count_leaves(tree[0])+
	  count_leaves(tree[1:])

	def is_leaf(item): return type(item)!= tuple

Flatten the tree	def enumerate_tree(tree):
	  if tree == ():
	  return ()
	  elif is_leaf(tree):
	  return (tree,)
	  else:
	  return enumerate_tree(tree[0])+
	  enumerate_tree(tree[1:])

Memoization	def make_cc_memo():
	  table = {}
	  def helper(a,d):
	  if (a,d) in table:
	  return table[(a,d)]
	  elif a < 0 or d == ():
	  return 0
	  elif a == 0:
	  return 1
	  else:
	  result = helper(a-d[0], d)
	  + helper(a, d[1:])
	  table[(a,d)] = result
	  return result
	  return helper