Variables:

```
# Variables are dynamicly typed
n = 0
print('n =', n)
>>> n = 0

n = "abc"
print('n =', n)
>>> n = abc

# Multiple assignments
n, m = 0, "abc"
n, m, z = 0.125, "abc", False

# Increment
n = n + 1 # good
n += 1    # good
n++       # bad

# None is null (absence of value)
n = 4
n = None
print("n =", n)
>>> n = None
```

If Statements:

```python
# If statements don't need parentheses
# or curly braces.
n = 1
if n > 2:
    n -= 1
elif n == 2:
    n *= 2
else:
    n += 2

# Parentheses needed for multi-line conditions.
# and = &&
# or  = ||
n, m = 1, 2
if ((n > 2 and
    n != m) or n == m):
    n += 1
```

Loops:

```python
n = 5
while n < 5:
    print(n)
    n += 1

# Looping from i = 0 to i = 4
for i in range(5):
    print(i)

# Looping from i = 2 to i = 5
for i in range(2, 6):
    print(i)

# Looping from i = 5 to i = 2
for i in range(5, 1, -1):
    print(i)
```

Math:

```
# Division is decimal by default
print(5 / 2)

# Double slash rounds down
print(5 // 2)

# CAREFUL: most languages round towards 0 by default
# So negative numbers will round down
print(-3 // 2)

# A workaround for rounding towards zero
# is to use decimal division and then convert to int.
print(int(-3 / 2))


# Modding is similar to most languages
print(10 % 3)

# Except for negative values
print(-10 % 3)

# To be consistent with other languages modulo
import math
from multiprocessing import heap
print(math.fmod(-10, 3))

# More math helpers
print(math.floor(3 / 2))
print(math.ceil(3 / 2))
print(math.sqrt(2))
print(math.pow(2, 3))

# Max / Min Int
float("inf")
float("-inf")

# Python numbers are infinite so they never overflow
print(math.pow(2, 200))
```

```python
# But still less than infinity
print(math.pow(2, 200) < float("inf"))
```

Arrays:

```python
# Arrays (called lists in python)
arr = [1, 2, 3]
print(arr)

# Can be used as a stack
arr.append(4)
arr.append(5)
print(arr)

arr.pop()
print(arr)

arr.insert(1, 7)
print(arr)

arr[0] = 0
arr[3] = 0
print(arr)

# Initialize arr of size n with default value of 1
n = 5
arr = [1] * n
print(arr)
print(len(arr))

# Careful: -1 is not out of bounds, it's the last value
arr = [1, 2, 3]
print(arr[-1])

# Indexing -2 is the second to last value, etc.
print(arr[-2])

# Sublists (aka slicing)
arr = [1, 2, 3, 4]
print(arr[1:3])

# Similar to for-loop ranges, last index is non-inclusive
print(arr[0:4])
```

```python
# But no out of bounds error
print(arr[0:10])

# Unpacking
a, b, c = [1, 2, 3]
print(a, b, c)

# Be careful though
# a, b = [1, 2, 3]

# Loop through arrays
nums = [1, 2, 3]

# Using index
for i in range(len(nums)):
    print(nums[i])

# Without index
for n in nums:
    print(n)

# With index and value
for i, n in enumerate(nums):
    print(i, n)

# Loop through multiple arrays simultaneously with unpacking
nums1 = [1, 3, 5]
nums2 = [2, 4, 6]
for n1, n2 in zip(nums1, nums2):
    print(n1, n2)

# Reverse
nums = [1, 2, 3]
nums.reverse()
print(nums)

# Sorting
arr = [5, 4, 7, 3, 8]
arr.sort()
print(arr)

arr.sort(reverse=True)
print(arr)
```

```python
arr = ["bob", "alice", "jane", "doe"]
arr.sort()
print(arr)

# Custom sort (by length of string)
arr.sort(key=lambda x: len(x))
print(arr)


# List comprehension
arr = [i for i in range(5)]
print(arr)

# 2-D lists
arr = [[0] * 4 for i in range(4)]
print(arr)
print(arr[0][0], arr[3][3])

# This won't work
# arr = [[0] * 4] * 4
```

Strings:

```python
# Strings are similar to arrays
s = "abc"
print(s[0:2])

# But they are immutable
# s[0] = "A"

# So this creates a new string
s += "def"
print(s)

# Valid numeric strings can be converted
print(int("123") + int("123"))

# And numbers can be converted to strings
print(str(123) + str(123))

# In rare cases you may need the ASCII value of a char
print(ord("a"))
print(ord("b"))

# Combine a list of strings (with an empty string delimitor)
strings = ["ab", "cd", "ef"]
print("".join(strings))
```

Queues:

```
# Queues (double ended queue)
from collections import deque

queue = deque()
queue.append(1)
queue.append(2)
print(queue)

queue.popleft()
print(queue)

queue.appendleft(1)
print(queue)

queue.pop()
print(queue)
```

Hashsets:

```
# HashSet
mySet = set()

mySet.add(1)
mySet.add(2)
print(mySet)
print(len(mySet))

print(1 in mySet)
print(2 in mySet)
print(3 in mySet)

mySet.remove(2)
print(2 in mySet)

# list to set
print(set([1, 2, 3]))

# Set comprehension
mySet = { i for i in range(5) }
print(mySet)
```

HashMaps:

```python
# HashMap (aka dict)
myMap = {}
myMap["alice"] = 88
myMap["bob"] = 77
print(myMap)
print(len(myMap))

myMap["alice"] = 80
print(myMap["alice"])

print("alice" in myMap)
myMap.pop("alice")
print("alice" in myMap)

myMap = { "alice": 90, "bob": 70 }
print(myMap)

# Dict comprehension
myMap = { i: 2*i for i in range(3) }
print(myMap)

# Looping through maps
myMap = { "alice": 90, "bob": 70 }
for key in myMap:
    print(key, myMap[key])

for val in myMap.values():
    print(val)

for key, val in myMap.items():
    print(key, val)
```

Tuples:

```python
# Tuples are like arrays but immutable
tup = (1, 2, 3)
print(tup)
print(tup[0])
print(tup[-1])

# Can't modify
# tup[0] = 0

# Can be used as key for hash map/set
myMap = { (1,2): 3 }
print(myMap[(1,2)])

mySet = set()
mySet.add((1, 2))
print((1, 2) in mySet)

# Lists can't be keys
# myMap[[3, 4]] = 5
```

Heaps:

```
import heapq

# under the hood are arrays
minHeap = []
heapq.heappush(minHeap, 3)
heapq.heappush(minHeap, 2)
heapq.heappush(minHeap, 4)

# Min is always at index 0
print(minHeap[0])

while len(minHeap):
    print(heapq.heappop(minHeap))

# No max heaps by default, work around is
# to use min heap and multiply by -1 when push & pop.
maxHeap = []
heapq.heappush(maxHeap, -3)
heapq.heappush(maxHeap, -2)
heapq.heappush(maxHeap, -4)

# Max is always at index 0
print(-1 * maxHeap[0])

while len(maxHeap):
    print(-1 * heapq.heappop(maxHeap))

# Build heap from initial values
arr = [2, 1, 8, 4, 5]
heapq.heapify(arr)
while arr:
    print(heapq.heappop(arr))
```

Functions:

```python
def myFunc(n, m):
    return n * m

print(myFunc(3, 4))

# Nested functions have access to outer variables
def outer(a, b):
    c = "c"

    def inner():
        return a + b + c
    return inner()

print(outer("a", "b"))

# Can modify objects but not reassign
# unless using nonlocal keyword
def double(arr, val):
    def helper():
        # Modifying array works
        for i, n in enumerate(arr):
            arr[i] *= 2

        # will only modify val in the helper scope
        # val *= 2

        # this will modify val outside helper scope
        nonlocal val
        val *= 2
    helper()
    print(arr, val)
```

```
nums = [1, 2]
val = 3
double(nums, val)
```

Classes:

```python
class MyClass:
    # Constructor
    def __init__(self, nums):
        # Create member variables
        self.nums = nums
        self.size = len(nums)

    # self key word required as param
    def getLength(self):
        return self.size

    def getDoubleLength(self):
        return 2 * self.getLength()

myObj = MyClass([1, 2, 3])
print(myObj.getLength())
print(myObj.getDoubleLength())
```