Basics Cheatsheet

Built-in

```
# primitives
x = 1            # integer
x = 1.2          # float
x = True         # bool
x = None         # null
x = float('inf')  # infinity (float)

# objects
x = [1, 2, 3] # list (Array)
x = (1, 2, 3) # tuple (Immutable Array)
x = {1: "a"}  # dict (HashMap)
x = {1, 2, 3} # set
x = "abc123"  # str (Immutable String)
```

Casting

```
n = 7
str(n)   # '7'
int('7') #  7

x = [1,2,3]
set(x)   # {1,2,3}
tuple(x) # (1,2,3)

s = {1,2,3}
list(s)  # [3,1,2]  (sets don't store order)
```

Basic Operations

```
a = [1] # array with the number 1 in it
b = [1]

a == b  # True, compares value
a is b  # False, compares memory location (exact comparison)

1 / 2    # .5 (division)
1 % 2    # 1 (remainder)
1 // 2   # 0 (division, rounding down)
2 ** 5   # 32 (power, 2^5)
```

Falsy Values

In Python, you can use anything as a boolean. Things that "feel false" like None, 0, and empty data structures evaluate to False.

```
a = None
b = []
c = [15]

if a:
    print('This does not print')
if b:
    print('This does not print')
if c:
    print('This prints')
```

Loops with `range`

```
for i in range(4):
    # 0 1 2 3

for i in range(1, 4):
    # 1 2 3

for i in range(1, 6, 2): # loop in steps of 2
    # 1 3 5

for i in range(3, -1, -1): # loop backwards
    # 3 2 1 0
```

Loops over Data Structures

```
arr = ["a", "b"]
for x in arr:
    # "a" "b"

hmap = {"a": 4, "b": 5}
for x in hmap:
    # "a" "b"
```

List Features

```
x = [1,2] + [3,4]  # [1,2,3,4]
x = [0, 7]*2       # [0,7,0,7], don't use this syntax for 2D arrays

x = [0,1,2,3]
x[2:]     # [2,3]
x[:2]     # [0,1]
x[1:4]    # [1,2,3]
x[3::-1]  # [3,2,1,0]
```

```
x[-1]     # 3

y = reversed(x)  # reversed array of x
x.reverse()      # reverses x in-place using no extra memory
sorted(x)        # sorted array of x
x.sort()         # sorts x in-place using no extra memory
```

List Comprehensions

Python has nice syntax for creating Arrays. Here's an example:

```
x = [2*n for n in range(4)]
#   [0, 2, 4, 6]

x = [a for a in [1,2,3,4,5,6] if a % 2 == 0]
#   [2, 4, 6]

# 2 x 3 matrix of zeros
x = [[0 for col in range(3)] for row in range(2)]
#  [[0, 0, 0],
#   [0, 0, 0]]
```

Here's the general form of list comprehension, and what it's actually doing:

```
x = [a*b   for a in A   for b in B   if a > 5]
# Is equivalent to:
x = []
for a in A:
    for b in B:
        if a > 5:
            x.append(a*b)
```

Generator Comprehensions

Generator Comprehensions are List Comprehensions, but they generate values lazily and can stop early. To do a generator comprehension, just use `()` instead of `[]`.

```
# stops early if finds True
any(SlowFn(i) for i in range(5))

# does not run the loop yet
computations = (SlowFn(i) for i in range(5))

# runs the loop - might stop early
for x in computations:
    if not x:
        break
```

Note that `()` can mean a tuple, or it can mean a generator. It just depends on context. You can think of Generator Comprehensions as being implemented exactly the same as List Comprehensions, but replacing the word `return` with `yield` (you don't have to know about this for an interview).

String Features

```
# you can use '' or "", there's no difference
x = 'abcde'
y = "abcde"
x[2] # 'c'

for letter in x:
    # "a" "b" "c" "d" "e"

x = 'this,is,awesome'
y = x.split(',')
print(y) # ['this', 'is', 'awesome']



x = ['this', 'is', 'awesome']
y = '!'.join(x)
print(y) # 'this!is!awesome'
```

```
# convert between character and unicode number
ord("a") # 97
chr(97) # 'a'
```

Set Features

```
x = {"a", "b", "c"}
y = {"c", "d", "e"}
y = x | y # merge sets, creating a new set {"a", "b", "c", "d", "e"}
```

Functions

You declare a function using the `def` keyword:

```
def my_function():
    # do things here

my_function() # runs the code in the function
```

All variables you declare inside a function in Python are local. In most cases you need to use `nonlocal` if you want to set variables outside a function, like below with `x` and `y`

```
x = 1
y = 1
z = [1]

def fn():
    nonlocal y
    y = 100      # global
    x = 100      # local
    z[0] = 100   # global (this would normally give an error. to avoid this, python refers to a more globally scoped
variable)
fn()
```

```
x    # 1
y    # 100
z[0] # 100
```

Anonymous Functions

You can also declare a function in-line, using the keyword `lambda`. This is just for convenience. These two statements are both the same function:

```
def fn(x,y):
    return x + y


lambda x,y: x + y
```

Boolean Operators

You can use the `any` function to check if any value is true, and the `all` to check if all values are true.

```
any([True, False, False]) # True
all([True, False, False]) # False


x = [1,2,3]
# checks if any value is equal to 3
any(True if val == 3 else False for val in x) # True
```

Ternary Operator

Most languages have a "Ternary operator" that gives you a value based on an if statement. Here's Python's:

```
0 if x == 5 else 1 # gives 0 if x is equal to 5, else gives 1

# Many other languages write this as (x == 5 ? 0 : 1)
```

Newlines

You can write something on multiple lines by escaping the newline, or just using parentheses.

```
x = 5 \
    + 10 \
    + 6

x = (
    5
    + 10
    + 6
)
```

Object Destructuring

You can assign multiple variables at the same time. This is especially useful for swapping variables. Here are a few examples:

```
# sets a=1, b=2
a,b = 1,2

# sets a=b, b=a, without needing a temporary variable
a,b = b,a

# sets a=1, b=2, c=3, d=4, e=[5, 6]
[a, b, [c, d], e] = [1, 2, [3, 4], [5, 6]]
```

Python Reference

Here's a refrence to the offical Python docs.

https://docs.python.org/3/library/index.html

The Built-in Functions and Built-in Types sections are the most useful parts to skim, although it's totally optional reading. The docs are not formatted in a very readable way.

Revision #9
Created 26 February 2024 22:04:44 by victor
Updated 22 October 2024 22:19:12 by victor