## Basics of Python

### Containers

Python includes several built-in container types: lists, dictionaries, sets, and tuples.

#### Lists

A list is the Python equivalent of an array, but is resizeable and can contain elements of different types:

xs = [3, 1, 2] # Create a list

print(xs, xs[2])

print(xs[-1]) # Negative indices count from the end of the list; prints "2"

xs[2] = 'foo' # Lists can contain elements of different types

print(xs)

xs.append('bar') # Add a new element to the end of the list

print(xs)

x = xs.pop() # Remove and return the last element of the list

print(x, xs)

As usual, you can find all the gory details about lists in the [documentation](https://docs.python.org/2/tutorial/datastructures.html#more-on-lists).

#### Slicing

In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as slicing:

nums = range(5) # range is a built-in function that creates a list of integers

print(nums) # Prints "[0, 1, 2, 3, 4]"

print(nums[2:4]) # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"

print(nums[2:]) # Get a slice from index 2 to the end; prints "[2, 3, 4]"

print(nums[:2]) # Get a slice from the start to index 2 (exclusive); prints "[0, 1]"

print(nums[:]) # Get a slice of the whole list; prints ["0, 1, 2, 3, 4]"

print(nums[:-1]) # Slice indices can be negative; prints ["0, 1, 2, 3]"

# nums[2:4] = [8, 9] # Assign a new sublist to a slice (this will fail!)

new\_nums = [i for i in nums]

new\_nums[2:4] = [8, 9] # Assign a new sublist to a slice

print(new\_nums) # Prints "[0, 1, 8, 9, 4]"

#### Loops

You can loop over the elements of a list like this:

animals = ['cat', 'dog', 'monkey']

for animal in animals:

print(animal)

If you want access to the index of each element within the body of a loop, use the built-in `enumerate` function:

animals = ['cat', 'dog', 'monkey']

for idx, animal in enumerate(animals):

print('#%d: %s' % (idx + 1, animal))

#### List comprehensions:

When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

nums = [0, 1, 2, 3, 4]

squares = []

for x in nums:

squares.append(x \*\* 2)

print(squares)

You can make this code simpler using a list comprehension:

nums = [0, 1, 2, 3, 4]

squares = [x \*\* 2 for x in nums]

print(squares)

List comprehensions can also contain conditions:

nums = [0, 1, 2, 3, 4]

even\_squares = [x \*\* 2 for x in nums if x % 2 == 0]

print(even\_squares)

#### Dictionaries

A dictionary stores (key, value) pairs, similar to a `Map` in Java or an object in Javascript. You can use it like this:

d = {'cat': 'cute', 'dog': 'furry'} # Create a new dictionary with some data

print(d['cat']) # Get an entry from a dictionary; prints "cute"

print('cat' in d) # Check if a dictionary has a given key; prints "True"

d['fish'] = 'wet' # Set an entry in a dictionary

print(d['fish']) # Prints "wet"

print(d['monkey']) # KeyError: 'monkey' not a key of d

print(d.get('monkey', 'N/A')) # Get an element with a default; prints "N/A"

print(d.get('fish', 'N/A')) # Get an element with a default; prints "wet"

del d['fish'] # Remove an element from a dictionary

print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"

You can find all you need to know about dictionaries in the **[documentation](https://docs.python.org/2/library/stdtypes.html#dict).**

It is easy to iterate over the keys in a dictionary:

d = {'person': 2, 'cat': 4, 'spider': 8}

for animal in d:

legs = d[animal]

print('A %s has %d legs' % (animal, legs))

If you want access to keys and their corresponding values, use the iteritems method:

d = {'person': 2, 'cat': 4, 'spider': 8}

# dict.iteritems() will fail in Python 3

for animal, legs in d.items():

print('A %s has %d legs' % (animal, legs))

Dictionary comprehensions: These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

nums = [0, 1, 2, 3, 4]

even\_num\_to\_square = {x: x \*\* 2 for x in nums if x % 2 == 0}

print(even\_num\_to\_square)

#### Sets

A set is an unordered collection of distinct elements. As a simple example, consider the following:

animals = {'cat', 'dog'}

print('cat' in animals ) # Check if an element is in a set; prints "True"

print('fish' in animals) # prints "False"

animals.add('fish') # Add an element to a set

print('fish' in animals)

print(len(animals)) # Number of elements in a set;

animals.add('cat') # Adding an element that is already in the set does nothing

print(len(animals))

animals.remove('cat') # Remove an element from a set

print(len(animals))

\_Loops\_: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

animals = {'cat', 'dog', 'fish'}

for idx, animal in enumerate(animals):

print('#%d: %s' % (idx + 1, animal))

# Prints "#1: fish", "#2: dog", "#3: cat"

Set comprehensions: Like lists and dictionaries, we can easily construct sets using set comprehensions:

from math import sqrt

print({int(sqrt(x)) for x in range(30)})

#### Tuples

A tuple is an (immutable) ordered list of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a trivial example:

d = {(x, x + 1): x for x in range(10)} # Create a dictionary with tuple keys

t = (5, 6) # Create a tuple

print(type(t))

print(d[t])

print(d[(1, 2)])

t[0] = 1

### Functions

Python functions are defined using the `def` keyword. For example:

def sign(x):

if x > 0:

return 'positive'

elif x < 0:

return 'negative'

else:

return 'zero'

for x in [-1, 0, 1]:

print(sign(x))

We will often define functions to take optional keyword arguments, like this:

def hello(name, loud=False):

if loud:

print('HELLO, %s' % name.upper())

else:

print('Hello, %s!' % name)

hello('Bob')

hello('Fred', loud=True)

### Unpacking Arguments

def my\_sum(my\_integers):

result = 0

for x in my\_integers:

result += x

return result

list\_of\_integers = [1, 2, 3]

print(my\_sum(list\_of\_integers))

def my\_sum(\*args):

result = 0

# Iterating over the Python args tuple

for x in args:

result += x

return result

print(my\_sum(1, 2, 3))

Note: The unpacking operator \* produces a a tuple, not a list. Both tuple and list are iterables, but tuple is immutable.

def concatenate(\*\*kwargs):

result = ""

# Iterating over the Python kwargs dictionary

for arg in kwargs.values():

result += arg

return result

print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))

# Create a sentence from a list of words.

def create\_sentence(\*\*words):

return ' '.join(words.values())

print(create\_sentence(a="Real", b="Python", c="Is", d="Great", e="!"))

### Classes

The syntax for defining classes in Python is straightforward:

class Greeter:

# Constructor

def \_\_init\_\_(self, name):

self.name = name # Create an instance variable

# Instance method

def greet(self, loud=False):

if loud:

print('HELLO, %s!' % self.name.upper())

else:

print('Hello, %s' % self.name)

g = Greeter('Fred') # Construct an instance of the Greeter class

g.greet() # Call an instance method; prints "Hello, Fred"

g.greet(loud=True) # Call an instance method; prints "HELLO, FRED!"