

DISCRETE STRUCTURES

Lab 1 Python Basics

Trần Hồng Tài

Abstract

Welcome to Python, a popular modern programming language that widely use in research and industry alike. In this Laboratory, we will introduce how to start with Python, basic features of Python via reprogramming common calculations and algorithms.

1 Python Installation

2 Python Tutorial

We will use the Python programming language for all assignments in this course. Python is a great general-purpose programming language; this section will serve as a quick crash course on the Python programming language. Also, any basic Python features can be found introduce [in this documentation](#). The content of this section is as follows:

2.1 Basic data types

Like most languages, Python has a number of basic types including integers, floats, booleans, and strings. These data types behave in ways that are familiar from other programming languages.

Numbers: Integers and floats work as you would expect from other languages:

```
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x)       # Prints "3"
print(x + 1)   # Addition; prints "4"
print(x - 1)   # Subtraction; prints "2"
print(x * 2)   # Multiplication; prints "6"
print(x ** 2)  # Exponentiation; prints "9"
x += 1
print(x)       # Prints "4"
x *= 2
print(x)       # Prints "8"
```

```
y = 2.5
print(type(y)) # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

Note: Unlike many languages, Python does not have unary increment ($x++$) or decrement ($x--$) operators. Python also has built-in types for complex numbers;

Booleans: Python implements all of the usual operators for Boolean logic, but uses English words (*and*, *or*, etc.) rather than symbols ($\&\&$, $\|\|$, etc.):

```
t = True
f = False
print(type(t)) # Prints "<class 'bool'>"
print(t and f) # Logical AND; prints "False"
print(t or f)  # Logical OR; prints "True"
print(not t)   # Logical NOT; prints "False"
print(t != f)  # Logical XOR; prints "True"
```

Strings: Python has great support for strings:

```
hello = 'hello'    # String literals can use single quotes
world = "world"    # or double quotes; it does not matter.
print(hello)       # Prints "hello"
print(len(hello))  # String length; prints "5"
hw = hello + ' ' + world # String concatenation
print(hw)          # prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12) # sprintf style string
                                     # formatting
print(hw12)        # prints "hello world 12"
```

String objects have a bunch of useful methods; for example:

```
s = "hello"
print(s.capitalize()) # Capitalize a string; prints "Hello"
print(s.upper())      # Convert a string to uppercase; prints "
                        HELLO"
print(s.rjust(7))     # Right-justify a string, padding with
                        spaces; prints "  hello"
print(s.center(7))    # Center a string, padding with spaces;
                        prints "  hello "
print(s.replace('l', '(ell)')) # Replace all instances of one
                                substring with another;
print(' world '.strip()) # Strip leading and trailing whitespace;
                        prints "world"
```

2.2 Containers

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

2.2.1 List

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

```
xs = [3, 1, 2]      # Create a list
print(xs, xs[2])    # Prints "[3, 1, 2] 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)           # Prints "[3, 1, 'foo']"
xs.append('bar')     # Add a new element to the end of the list
print(xs)           # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()        # Remove and return the last element of the list
print(x, xs)        # Prints "bar [3, 1, 'foo']"
```

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

```
nums = list(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
print(nums)              # Prints "[0, 1, 8, 9, 4]"
```

We will see slicing again in the context of numpy arrays. **Loops:** You can loop over the elements of a list like this:

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
# Prints "cat", "dog", "monkey", each on its own line.
```

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))
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

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)          # Prints [0, 1, 4, 9, 16]
```

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)    # Prints [0, 1, 4, 9, 16]
```

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)    # Prints "[0, 4, 16]"
```

2.2.2 Dictionary

A dictionary stores pairs of (*key,value*). 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"
```

Loops: 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))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8
    legs"
```

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

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8
    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) # Prints "{0: 0, 2: 4, 4: 16}"
```

2.2.3 Set

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) # Prints "True"
print(len(animals))      # Number of elements in a set; prints "3"
animals.add('cat')       # Adding an element that is already in
                        # the set does nothing
print(len(animals))      # Prints "3"
animals.remove('cat')    # Remove an element from a set
print(len(animals))      # Prints "2"
```

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
nums = {int(sqrt(x)) for x in range(30)}
print(nums) # Prints "{0, 1, 2, 3, 4, 5}"
```

2.2.4 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))                         # Prints "<class 'tuple'>"
print(d[t])                           # Prints "5"
print(d[(1, 2)])                      # Prints "1"
```

2.3 Function

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))
    # Prints "negative", "zero", "positive"
```

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') # Prints "Hello, Bob"
hello('Fred', loud=True) # Prints "HELLO, FRED!"
```

2.4 Class

The syntax for defining classes in Python is straightforward:

```
class Greeter(object):

    # 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!"
```