USC Viterbi School of Engineering

PYTHON FUNDAMENTALS

EE 541 – UNIT 2A

DR. BRANDON FRANZKE

Spring 2023

PYTHON PROGRAMS

- Program: a sequence of definitions and commands
  - Evaluate definitions
  - Interpreter executes commands
- Commands are statements that instruct the interpreter to do something
- Can run commands interactively (in a shell) or stored in file and read into shell
- Programs manipulate data objects – stored in memory
- Objects have a type that defines valid operations
  - Scalar ("atomic") vs non-scalar (internal structure)

INDENTATION DEFINES CODE BLOCKS

- Very important and a common cause of errors
  - especially if deeply nested
  - use 2 or 4 spaces (instead of Tab)

```
x = float(input("Enter a number x: “))
y = float(input("Enter a number y: "))
if x == y:
    print("x equals y")
    if y != 0:
        print("x / y =", x/y)
elif x < y:
    print("x is smaller")
else:
    print("y is smaller")
```

SOURCES OF ERRORS

- Syntax errors
  - Very common and easy to resolve
- Static semantic errors
  - Checks before program execution
  - Lead to unpredictable behavior
- **No semantic errors but implementation differs from programmer intent**
  - Program crashes
  - Program runs forever
  - Program gives a different answer than expected

DATA TYPES

SCALAR OBJECT TYPES

- `int` represent integers: `5, 0, -1`
- `float` represent real numbers: `3.14, 5.00, 0.11`
- `bool` represent Boolean values, `True` and `False`
- `NoneType` Python "special" type, only one possible value: `None`
- Cast to change type: `int(3.0), float(3)`
- Use `type()` to get object type

```
>>> type(3.14)     # float
>>> type(-1)       # int
```

SIMPLE OPERATIONS AND ASSIGNMENT

- Sum (`+`), Difference (`-`), product (`*`), division (`/` and `//`)
- Modulus `%`
- Power (exponent) `**`
- Use parentheses to dictate order of operations, else

  `** > * > / > +, -`

- Use equal sign to assign value to name (a “variable”)

```
>>> pi = 3.14159
>>> pi_almost = 22/7
```

STRING TYPE

- Contain letters, special characters, spaces, digits
- Enclose in single quotes or quotation marks
  - Personal preference, but be consistent
- Concatenate with "+"

```
>>> what = "dog"
>>> pet = "good" + what              # gooddog
>>> mypet = "good" + " " + what    # good dog
```

INPUT + OUTPUT (I/O)

- Command print() to output to console

```
>>> x = 3
>>> print(x)
>>> print("a message to user")
>>> print("error ", errcode, "occurred")
```

- Command `input()` to get string from user
  - **Always gives a string**

```
>>> your_num = input("Pick a number, any number")
>>> your_num = int(your_num)
```

SCALAR OBJECT COMPARISONS

- `i` and `j` are variable names
- Comparisons produce a Boolean
  - `i > j`
  - `i >= j`
  - `i < j`
  - `i <= j`
  - `i == j    # test equality`
    `           # (True if i and j are "same")`
  - `i != j    # test inequality`
    `           # (True if i and j are not same)`

BOOLEAN LOGIC OPERATIONS

- `a` and `b` are Boolean variables names
  - `not a`
  - `a and b`
  - `a or b`

```
>>> num = input("Input an integer")
>>> num = int(num)
>>> print("Your number is less than 10", num < 10)
>>> print (num <= 10 and num >= 10)
       # same as num == 10
```

STRING MANIPULATION

- Case sensitive
- Compare with usual operators: ==, >, <=, etc.
- retrieve length with `len()`

```
>>> len("EE541")  # 5
```

- index with square brackets and position (0-index)
  - `>>> s = "EE541"`
  - `>>> s[0]     # "E"`
  - `>>> s[2]     # "5"`
  - `>>> s[5]     # Error, OutOfBounds`
  - `>>> s[-1]    # 1`

STRING SLICING (VERY IMPORTANT CONCEPT)

- Slice string with `[start:stop:step]`
  - default step = 1, `[start:stop]`

- Omit number to indicate “start” or “end” respectively

```
>>> s = "EE541 Lecture2"
>>> s[3:7]      # "41 L"
                # equivalent to [3:7:1]
>>> s[3:7:2]    # "4 "
>>> s[::]       # "EE541 Lecture2",
                # equivalent [0:len(s),1]
>>> s[::-1]     # "2erutceL 145EE"
                # equivalent [-1:-(len(s)+1):-1]
>>> s[4:1:-2]   # "15"
```

CONTROL STRUCTURES

USC Viterbi School of Engineering

BRANCHING AND CONDITIONAL FLOW CONTROL

```
if <condition>:
  <expression>
  <expression>
```

```
if <condition>:
  <expression>
  <expression>
else:
  <expression>
  <expression>
```

```
if <condition>:
  <expression>
  <expression>
elif
<condition>:
  <expression>
  <expression>
else:
  <expression>
  <expression>
```

Interpreter *casts* `<condition>` to `True` or `False`

LOOPS AND ITERATION CONTROL

```
while <condition>:
    <expression>
    <expression>
```

- Interpreter casts <condition> to True or False
- If <condition> True, ***run*** nested block
- If <condition> False, ***skip*** nested block
- Repeat (forever) until <condition> is False

WHILE LOOP EXAMPLE

Iterate through a sequence

```
n = 0
while n < 10:
    print(n)
    n = n + 1

# more common alternative, for loop
for n in range(10):
    print(n)
```

ITERATION WITH FOR LOOPS

```
for <variable> in <iterable>:
    <expression>
    <expression>
```

- `<variable>` get a new value each time through the loop
- first iteration: `<variable> = next(<iterable>)`
- second iteration: `<variable> = next(<iterable>)`
- … etc

RANGE(START, STOP, STEP) TO DEFINE ITERATOR

default values: start = 0, step = 1

```
cumsum = 0
for i in range(5, 10):
    cumsum += i
print(cumsum)              # 5 + 6 + 7 + 8 + 9

cumsum = 0
for i in range(5, 10, 2):
    cumsum += i
print(cumsum)              # 5 + 7 + 9
```

USE BREAK TO IMMEDIATELY EXIT A LOOP

- exit whatever loop the command is in
- skip all remaining expressions in the block
- only exists **INNERMOST** loop

```
while <condition_1>:
  while <condition_2>:
    <expression_a>
    break
    <expression_b>
  <expression_c>
```

Early loop termination

```
cumsum = 0
for i in range(5, 100, 2):
  cumsum += i
  if cumsum > 25:
    break
    cumsum = 0
print(cumsum)
```

EXAMPLE: CUBE ROOT

```
cube = 27
epsilon = 0.01
num_guess = 0
low = 0
high = cube
guess = (high + low) / 2.0
while abs(guess**3 - cube) >= epsilon:
  if guess**3 < cube:
    low = guess
  else:
    high = guess
  guess = (high + low) / 2.0
  num_guess += 1
print("Guess number", guess_num, "is", guess, "and is 
close to cube root of", cube)
```

USC Viterbi School of Engineering

ABSTRACTION

VARIABLES

- Use variables liberally
  - Simplify code
  - Reuse names instead of expressions
  - Easier to change or modularize program

- Re-assign a new value to overwrite existing
  - Value may still exist in memory
  - Requires “cleanup” or “garbage collection”

DECOMPOSITION – ABSTRACTION WITH FUNCTIONS

- functions are
  - reusable pieces/blocks of code
  - mechanism to achieve **decomposition** and **abstraction**

- functions are not *run* until “called” or “invoked”

- function elements:
  - has a **name**
  - has 0 or more **parameters**
  - has a **docstring** (optional, describe function)
  - has a **body**
  - **returns** something (default is None if no return statement)

WRITING FUNCTIONS

```
def is_odd(k):
    """
    Input: k, a positive int
    Returns True if k is odd, otherwise False
    """
    print("inside body of is_odd")
    return k % 2 == 1

>>> is_odd(15)
```

FUNCTIONS CREATE SCOPE

- Formal parameter is the name in the argument list: `k`
- Actual parameter is the value *passed in*: `15`
- Entering a function creates a new scope (frame/ environment)
- Scope controls visibility of name to object map

```
def inc(x):
  x = x+ 1
  print(“in f(x): x = ”, x)
  return x

x = 5
z = inc(x)
```

SCOPE EXAMPLE

- Inside a function, **can access** variable defined outside
- Inside a function, **cannot modify** variable defined outside
  - requires global keyword (generally not good)

```
def f(y):
   x = 1
   x += 1
   print(x)

x = 5
f(x)
print(x)
```

```
def g(y):
   print(x)

print(x+1)

x = 5
g(x)
print(x)
```

```
def h(y):
   x += 1

x = 5
g(x)     # Error
print(x)
```

TEST YOUR UNDSTANDING: SCOPE

```
def g(x):
    def h():
        x = "abc"
    x = x + 1
    print("g: x =", x)
    h()
     return x

x = 3
z = g(x)
```

USC Viterbi School of Engineering

FUNCTIONS AS VARIABLES (ARGUMENTS)

```
def func_a():
  print("inside func_a")
def func_b(x):
  print("inside func_b")
  return x
def func_c(y):
  print("inside func_c")
  return y()


print(func_a())          # None
print(4 + func_b(3))     # 7
print(func_c(func_a))    # None
```

DATA STRUCTURES

TUPLES

- Tuples are a **compound** data type
- Ordered sequence of elements, can mix types
- **cannot change** element values, **immutable**
- create with parentheses: `()`

```
t1=()         # empty tuple        t[1:2]      # (52, )
t=("EE", 52, True)                 t[1:3]      # (52, True)
t[0]          # "EE"               len(t)      # 3
t+("a",6)   # new tuple            t[1] = 51   # Error
```

COMMON TUPLE PATTERNS

- Swap variables

```
x = y
y = x
```

```
temp = x
x = y
y = temp
```

```
(x, y) = (y, x)
```

- Return more than one value from a function

```
def div_q_and_r(x, y):
    q = x // y              # integer division
    r = x % y
    return (q, r)
(quot, rem) = div_q_and_r(6, 8);
```

EXAMPLE: MANIPULATING TUPLES

```
def extract_data(aTuple):
   nums = ()
   words = ()
   for t in aTuple:
      nums = nums + (t[0], )
      if t[1] not in words:
         words = words + (t[1], )
   (min_n, max_n) = (min(nums), max(nums))
   unique_words = len(words)
   return (min_n, max_n, unique_words)
```

aTuple: ((#,S), (#,S), (#,S), …)

nums: (#, #, #, …)

words: (S, S, S, …)

USC Viterbi School of Engineering

LISTS

- Lists are a **compound** data type
- Ordered sequence of elements, *usually* all same type
- **can change** element values, **mutable**
- create with square brackets: `[]`

```
L1 = []    # empty list
L = [4, 2, True, "abc", [1, 2]]
len(L)     # 5
L[1]+ 5    # 7
```

```
L[4]           # [1, 2]
L[5]           # Error
idx = 2
L[idx] = "cd"
```

COMMON LIST PATTERNS

```
total = 0
for k in range(len(L)):
  total += L[k]
print(total)
```

```
total = 0
for val in L:
  total += val
print(total)
```

- List elements indexed `0` to `len(L) - 1`
- `range(n)` gives list `0` to `n-1`

LIST OPERATIONS: LENGTHEN

- Add to end
  - `L.append(element)`
  - **mutates** the list object
  - `L.append()` is called a method

- Concatenate

```
L1 = [1, 2, 3]
L2 = [4, 5, 6]
L3 = L1 + L2          # unchanged L1 and L2
                      # len(L3) = 6
L1.extend([2, 1])     # mutated L1
                      # len(L1) = 5
```

LIST OPERATIONS: REDUCE

- Delete element at index
  - `del(L[index])`
- Remove the last element
  - `L.pop()`
  - `last_el = L.pop()    # returns the element`
- Remove an element
  - `L.remove(element)`
  - removes first occurrence
  - error if not in list

These all mutate the list

LIST TO STRING AND BACK AGAIN

- String to list, elements = single character strings (i.e., *letters*)

```
list(s)
list("EE 541")      # ["E", "E", " ", "5", "4", "1"]
```

- String to list, break at character (default, space)

```
s.split(" ")
"EE 541".split(" ")          # ["EE", "541"]
"A_snake_string".split("_") # ["A", "snake", "string"]
```

- List to string
  - `"".join()`
  - `"glue"` character in quotes put between each element
  - `"_".join(["a", "b", "c"])     # "a_b_c"`

USC Viterbi School of Engineering

DOCUMENTATION: MORE LIST OPERATIONS

- `sort()` / `sorted()`
- `reverse()`
- `count()`
- `index()`
- `clear()`

LISTS IN MEMORY

- lists are mutable
- behave differently than immutable types
- an object in computer memory
- variable names *points* to object
- mutation affects **every** variable pointing to that object
- keep in mind: *side effects*

USC Viterbi School of Engineering

ALIASES AND SIDE-EFFECTS

```
a = 1
b = a
print(a)                   # 1
print(b)                   # 1

warm = [“red”, “yellow”, “orange”]
hot = warm
hot.append(“pink”)    # beware, side effect
print(hot)                 # […, “pink”]
print(warm)                # […, “pink”]
```

CLONE TO AVOID SIDE-EFFECTS

```
cool = ["blue", "purple", "green"]
chill = cool[:]                    # clone
chill.append("black")
print(chill)                       # […, "black"]
print(cool)                        # […, "green"]
```

Clone creates a copy -

double memory but separate access and mutation

EXAMPLE: SORT AND MUTATION

- `sort()` **mutates** the list - returns nothing

```
warm = ["red", "yellow", "orange"]
sortedwarm = warm.sort()
print(warm)          # ["orange", "red", "yellow"]
print(sortedwarm)    # None
```

- `sorted()` **does not mutates** the list - returns sorted list

```
cool = ["blue", "purple", "green"]
sortedcool = sorted(cool)
print(cool)          # ["blue", "purple", "green"]
print(sortedcool)    # ["blue", "green", "purple"]
```

LISTS: THE TRICKY BITS

- Lists can contain lists – called “nested”
  - side effects still possible after mutation

- Avoid mutating a list as you iterate

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)
L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups(L1, L2)
    # L1 = [2, 3, 4]
```

```
def remove_dups(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

DICTIONARIES

- A dictionary is key-value map (a *hash*)

list

0	Element 1
1	Element 2
2	Element 3
3	Element 4
…	…

dictionary

Key 1	Value 1
Key 2	Value 2
Key 3	Value 3
Key 4	Value 4
…	…

CREATE A DICTIONARY

```
my_dict = {}                # empty dict
grades = {"John": "A", "Beth": "B+", "Mary": "A",
  "Bill": "B"}

grades["Beth"]              # "B+"
grades["Sam"]               # KeyError

grades["Sue"] = "A-"
"John" in grades            # True
"Dan" in grades             # False

del(grades["Mary"])
```

DICTIONARY KEYS AND VALUES

- Values can be any type (mutable, immutable, NoneType)
  - Duplicates OK
  - Can nest dictionaries, lists, tuples
- Keys must be unique
  - Immutable types only – not quite, but good enough for now
    - int, float, string, tuple, bool
  - Careful with float keys

```
grades = {"John": "A", "Beth": "B+", "Mary": "A",
  "Bill": "B"}

grades.keys()          # ["Mary", "Beth", "John", "Bill"]

grades.values()        # ["B", "A", "A", "B+"]
```

OBJECTS AND CLASSES

PYTHON OBJECT ORIENTED (OO)

- Python supports many datatypes
- Each type is an **Object**, every object has:
  - a type
  - an internal representation (primitive vs. composite)
  - set of procedures to interact with the object
- An object is an **instance** of a type
  - `5678` is an instance of an int
  - "`Bob`" is an instance of a string

In Python: *EVERYTHING IS AN OBJECT*

OBJECT-ORIENTED ADVANTAGES

- Bundle data into packages
  - include procedures to interact through well-defined interfaces
- Modular development
  - implement and test each module behaviors separately
  - reduced complexity
- Encourage code reuse
  - many Python modules define new classes
  - Each class has a separate environment
    - no function or variable name collisions
  - Subclasses inherit from classes to redefine or extend a class

OBJECTS ARE DATA ABSTRACTIONS

The abstraction captures:

1. an *Internal Representation*

   as data attributes

2. an *Interface*

   procedures to interact with the object – called methods

   define behavior but hide implementation

OO CONSIDERATIONS

- Distinguish
  - create a class
  - create an instance (of a class)
- Create a class
  - define the class name
  - define class attributes
- Create an instance
  - create a new instance of an object
  - commanding operations on the instance
  - *e.g.*, `L = [1, 2]` and `len(L)`

CLASSES DEFINE NEW DATATYPES

```
class Point(object):
    # define attributes here
    # …
```

- object means Point is a Python object
  - it **inherits** all the attributes
  - say: Point is a subclass of object
  - object is a superclass of Point

- Indent gives bounds of **class definition**

TWO TYPES OF CLASS ATTRIBUTES

- Attributes are data a procedures that “**belong**” to the class

1. members (data attributes) – *WHAT IS the object*

   other objects or data that make up the class

2. methods (procedural attributes) – *HOW TO use the object*

   functions that work only on this class

   define available interactions

CREATE A CLASS DEFINITION AND THEN INSTANCES

```
class Point(object):
  def __init__(self, x, y): # self always first argument
    self.x = x              # data attributes
                            # "instance variables"
    self.y = y

p = Point(3, 4)             # no "self", it's automatic
origin = Point(0, 0)
print(p.x)                  # "." operator
                            # to access attributes
print(origin.x)
```

METHODS PROVIDE THE INTERFACE

```
class Point(object):
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def distance(self, from):   # self always first argument
    x_diff_sq = (self.x - from.x)**2
    y_diff_sq = (self.y - from.y)**2
    return (x_diff_sq + y_diff_sq)**0.5
```

USING CLASS METHODS

```
p = Point(3, 4)
zero = Point(0, 0)
print(p.distance(zero))
```

```
p = Point(3, 4)
zero = Point(0, 0)
print(Point.distance(p, zero))
```

CLASSES HELPER METHODS

- `__str__()`
    - provide an **informative** print representation
    - default representation is **uninformative**
        - `<__main__.Point object at 0x7fa918510488>`
    - you can define the data and format

- Special operators
    - `__add__(self, other)        # self + other`
    - `__sub__(self, other)        # self - other`
    - `__eq__(self, other)         # self == other`
    - `__len__(self)               # len(self)`
    - …

GETTERS AND SETTERS: ACCESS CLASS DATA

```
class Animal(object):
  def __init__(self, age):           # constructor
    self.age = age
    self.name = None
  def get_age(self):                 # getter for age
    return self.age
  def get_name(self):                # getter for name
    return self.name
  def set_age(self, newage):         # setter for age
    self.age = newage
  def set_name(self, newname=“”):  # setter for name
    self.name = newname
  def __str__(self):                 # print format
    return "animal:" + str(self.name) + ":" +
str(self.age)
```

USE GETTERS AND SETTERS OUTSIDE CLASS

```
class Animal(object):
  def __init__(self, age_years):   # constructor
    self.age_years = age
    self.name = None
  def get_age(self):               # getter for age
    return self.age_years
  def get_name(self):              # getter for name
    return self.name
  def set_age(self, newage):       # setter for age
    self.age_years = newage
  def set_name(self, newname=""):  # setter for name
    self.name = newname
  def __str__(self):               # print format
    return "animal:" + str(self.name) + ":" +
str(self.age)
```

CLASS INHERITANCE

```
class Cat(Animal):
  def speak(self):            # new method
    print("meow")
  def __str__(self):          # override existing method
    return "cat:" + str(self.name) + ":" + str(self.age)
```

- A subclass can:
  - add new functionality (new methods, new data attributes)
  - override existing methods or data
- Use **parent** class method if not explicit in subclass, e.g., `__init__()`
  - recurse hierarchy, use first match

EXAMPLE: CLASS INHERITANCE

```
class Person(Animal):
  def __init__(self, name, age):
    Animal.__init__(self, age)          # Animal constructor
    self.set_name(name)                 # setter to Animal data
    self.friends = []
  def get_friends(self):
    return self.friends
  def add_friend(self, fname):
    if fname not in self.friends:
      self.friends.append(fname)
  def speak(self):
    print("hello")
  def age_diff(self, other):
    diff = self.age - other.age
    print(abs(diff), "year difference")
  def __str__(self):
    return "person:" + str(self.name) + ":" + str(self.age)
```

EXAMPLE: CLASS INHERITANCE

```
import random
class Student(Person):
  def __init__(self, name, age, Major=None):
    Person.__init__(self, name, age)
    self.major = major
  def change_major(self, newmajor):
    self.major = newmajor
  def speak(self):
    r = random.random()    # gives r in [0, 1]
    if r < 0.25:
      print("sleeping, come back later")
    elif r < 0.75:          # compare priority with r < 0.25
      print("studying, come back later")
    else:                   # r >= 0.75
      print("relaxing, join me")
  def __str__(self):
    return "student:" + str(self.name) + ":" + str(self.age) +
":" + str(self.major)
```

CLASS VARIABLES (SHARED ACROSS INSTANCES)

```
class Rabbit(Animal):
  tag = 1                           # class variable
                                    # unique id feature
  def __init__(self, age, parent1=None, parent2=None):
    Animal.__init__(self, age)
    self.parent1 = parent1
    self.parent2 = parent2
    self.rid = Rabbit.tag           # access class variable
    Rabbit.tag += 1                 # increment for next id
```

EXAMPLE: SHARED ID COUNTER

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
    def get_rid(self):
        return str(self.rid).zfill(3)            # pad with "0"
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
```

CREATE AN INTERFACE FOR STANDARD OPERATIONS

```
class Rabbit(Animal):
  […]
  __add__(self, other):        # offspring = rP + rM
    # return a new object of same type
    return Rabbit(0, self, other)       # age, p1, p2
  __eq__(self, other):         # DON'T compare directly
    parent_same = self.parent1.rid == other.parent1.rid \
      and self.parent2.rid == other.parent2.rid
    parent_swap = self.parent2.rid == other.parent1.rid \
      and self.parent1.rid == other.parent2.rid
    return parent_same or parent_swap
```