Programming in Python

MMM001 - Data Engineering

https://github.com/chbrandt/MMM001

Carlos H Brandt

c.brandt@jacobs-university.de

Table of Contents

(Some) built-in functions

- print (<arguments>)
  - Print arguments to stdout
- id (<object>)
  - Return "in-memory" identifier of object
- type (<object>)
  - Return type of object
- len (<sequence>)
  - Return length of sequence
- input (<message>)
  - Print message to stdout and read input from stdin
- range (<[start=0,] stop [,step=1]>)
  - Generate numbers in a range

Functions

- A function is a block of *reusable* code that accomplish a specific task
- Functions is Python may have zero, one or more arguments
  - Arguments may have default value
- Functions is Python may return zero, one or more values
- Functions can be *pure* or *non-pure*
  - *Pure* functions *do **not** change* their input arguments
  - *Non-pure* function *do change* their input arguments
- Functions may be imported or defined

```
def a_function(arg1, arg2='hello'):
    """
    A "Hello-World" function
    """
    return arg2 + ", " + arg1
```

Variables

- Variables are containers for storing data values
  - It is a name (or identifier), in a scope, pointing to a value with a type
  - I.e., names pointing to objects in memory (i.e., name binding)
- Python has no command for declaring a variable,
  - They are created the moment you first assign a value to it
- Variables can be of any type (integer, string, etc.)
  - and can even change their type after they have been set (dynamic typing)
- Variable names are case-sensitive:
  - can only contain alphanumeric characters and underscores
  - must start with a letter or the underscore character

```
>>> valid_name = 'ok'
```

```
>>> _also_valid = 'ok'
```

```
>>> 1_invalid = 'not ok'
```

Variables

- Python allows you to assign multiple values to multiple variables in one line

```
>>> a, b, c = 'one', 2, True
```

- And you can assign the same value to multiple variables in one line

```
>>> a = b = c = 10
```

- Variables created in the **top-level** of the code are called **global**
- **Global variables** can be **used anywhere**, inside and outside of functions
  - To modify a global variable inside a function, use the keyword *'global'*
  - To modify a non-local variable inside a function, use the keyword *'nonlocal'*

Reserved keywords

- **and**
- as
- assert
- **break**
- class
- **continue**
- **def**
- del
- **elif**
- **else**
- except
- **False**
- finally
- **for**
- **from**
- global
- **if**
- **import**
- **in**
- **is**
- lambda
- **None**
- nonlocal
- **not**
- **or**
- pass
- raise
- **return**
- **True**
- try
- while
- with
- yield

Data types

- Data in Python is represented by objects or their relations.
- A data type defines the kind of operations that make sense to a value and, internally, how it is structured in memory.
- Data types may be grouped into **numerics**, **sequences**, **sets** and **mappings**, And then further into **mutable** and **immutable** types:
  - Immutable: cannot be changed
    - bool, int, float, complex, str, tuple, frozenset, bytes
  - Mutable: content can be changed
    - list, dict, set, array, bytearray

Data structures

- Data structures are collections of data values organized in some particular way so to provide efficient access or storage.
- Among the built-in data types Python provides some basic data structures:
  - Dictionaries
  - Lists, tuples
  - Sets, frozen-sets
  - Arrays, byte-arrays

Assigning and using objects

- Numbers:

```
# integers:
>>> a = 9
>>> b = 0xf      # hexadecimal
>>> c = 0o7      # octal

# floats:
>>> a = 2.37
>>> b = -1.23

# complex:
>>> a = 10 + 2.3j
```

```
>>> 1 + 2.7
>>> 1j + 3
>>> 10 / 3
>>> 2*3
>>> 2**3
>>> 0xf - 15
>>> 3.1415 * 2
```

Assigning and using objects

- Booleans and the Null object:

```
>>> a = True
>>> b = False
>>> c = None
```

```
>>> True * False
>>> False - True
>>> True or None
>>> False or None
>>> True and False
```

Defining/assigning objects

- Strings:

```
>>> a = 'a string in single quotes'
>>> b = "a string in double quotes"
>>> c = "use 'quotes' inside quotes"
>>> d = 'and "vice-versa"'
>>> e = """triply quoted definition,
it allows line breaks"""
>>> f = '''usually used on docstrings'''
>>> g = '1000'
>>> h = 'True'
>>> i = 'None'
```

Assigning and using objects

- Tuples and Lists:

```
# Lists
>>> a = [1, 2, 3]
>>> b = ["heterogeneous", 1, 2.7, "bla"]
>>> c = ["also nested", [1, 2, 3]]

# Tuples are like list, but immutable
>>> a = ('a tuple', 1j)
>>> b = ('and mix', [1, 2, 3])
```

```
# Lists
>>> a = [1, 2, 3]
>>> a[0]
>>> a[1:2]
>>> a.append('yay')

# Tuples are like list, but immutable
>>> a = ('a tuple', 1j)
>>> a[0]
>>> a.append(10)
```

Assigning and using objects

- Sets and Dictionaries

```
# Sets
>>> a = set()
>>> b = set( [1, 2, 2, 2, 3])

# Dictionaries
>>> a = {'some': 1, 'another': 100}
```

```
# Sets
>>> a = set()
>>> a.add(1)
>>> a.add(1)

# Dictionaries
>>> a = {}
>>> a['a key'] = 'a value'
>>> del a['a key']
```

Type conversion (casting)

- Integer: int(<object>)
- Float: float(<object>)
- Complex: complex(<object>)
- String: str(<object>)
- Boolean: bool(<object>)
- List: list(<object>)
- Tuple: tuple(<object>)
- Set: set(<object>)

Implicit conversion will happen if allowed (by the operation) and will typically convert to a higher order type (e.g., integer to float)

Operators

- Assignment: =
- Addition: +
- Multiplication: *
- Subtraction: -
- Division: /
- Floor: //
- Power: **
- Modulus: %
- Execution order: ( )

- Less-then: <
- Greater-then: >
- Less-or-equal: <=
- Greater-or-equal: >=
- Equal: ==
- Not-equal: !=

- and
- or
- not
- in

References

- Built-in Functions: https://docs.python.org/3/library/functions.html
- Keywords: https://www.w3schools.com/python/python_ref_keywords.asp
- Python Data Model: https://docs.python.org/3/reference/datamodel.html
- Python Standard Types: https://docs.python.org/3/library/stdtypes.html
- Wikibooks: https://en.wikibooks.org/wiki/Python_Programming/Operators

Laboratory - Mutables and Immutables

```
# A number
>>> a = 1
>>> b = 1
>>> id(a)
>>> id(b)
```

```
# A string
>>> a = 'hi'
>>> b = 'hi'
>>> id(a)
>>> id(b)
```

```
# Tuples
>>> a = (1, 'hi')
>>> b = (1, 'hi')
>>> id(a)
>>> id(b)
```

```
# Lists
>>> a = [1, 'hi']
>>> b = [1, 'hi']
>>> id(a)
>>> id(b)
```

Q: What happened in each case? Are the id's the same? Why?

Laboratory - Mutables and Immutables

```
# Change a number?
>>> a = 1
>>> b = a
>>> print(a, b)
>>> b = 2
>>> print(a, b)
```

```
# Change a list
>>> a = [1, 'hi']
>>> b = a
>>> print(a, b)
>>> b[0] = 'yay'
>>> print(a)
```

```
# Change a tuple?
>>> a = (1, 'hi')
>>> b = a
>>> id(a)
>>> id(b)
>>> b[0] = 'yay'
```

Q: What happened in each case? Did all succeed; if yes, was the result expected, and if not, why?

Laboratory - Conversion

Exercise: do the following conversions, some will work, others not. Why?

```
>>> int( 9.8 )
>>> float( 4 )
>>> float( "3.8" )
>>> int( "3.8" )
>>> int( "3" )
>>> float( "3" )
>>> int( True )
>>> int( False )
```

```
>>> bool( 0 )
>>> bool( 1 )
>>> bool( -1 )
>>> bool( "" )
>>> bool( " " )
>>> bool( [ ] )
>>> bool( None )
>>> bool( [False] )
```

```
>>> str( 0 )
>>> str( None )
>>> str( -1 )
>>> str( False )
>>> str( [1, 2, 3] )
```

Laboratory - Conversion

Exercise: do the following conversions, some will work, others not.

```
>>> list( 1 )
>>> list( [1] )
>>> list( (1) )
>>> list( (1,) )
```

```
>>> list( (1,2,3) )
>>> list( 'abc' )
>>> tuple( [1,2,3] )
>>> tuple( 'abc' )
```

```
>>> set( [1,2,3] )
>>> set( [1,2,2,3] )
>>> set( 'abc' )
>>> list( set( [1,1,1] ) )
```

Q: What happened in each case? Are the results what was expected?

Laboratory - Operators

Exercise: do the following operations:

```
>>> 1 + 1
>>> 1 + True
>>> True + False
>>> True * False
>>> True / False
>>> 0 and 1
>>> 0 or 1
```

```
>>> "a" in ["a", "b", "c"]
>>> "a" in "abc"
>>> 1 not in [1, 2, 3]
>>> not 1 in [1, 2, 3]

>>> x = None
>>> x is None
```

```
>>> 2**3
>>> 10 * 2**3
>>> (10 * 2)**3
>>> 10 / 3
>>> 10 // 3
>>> 10 % 3
>>> not True
```

Q: What happened in each case? Are the results what was expected?

Laboratory - Data structures

```
>>> L = [1, 2, 3]
>>> L2 = L
>>> L3 = L[:]
>>> L2.append(None)
>>> print(L)
>>> print(L3)
>>> L3.pop()
>>> print(L3)
>>> print(L)
```

```
>>> D = {1:"a", "b":2}
>>> D[None] = "hm?"
>>> print(D)
>>> D2 = {None: "ok'..."}
>>> D2.update({'z': True})
>>> D.update(D2)
>>> del D2['z']
>>> print(D)
>>> print(D2)
```

```
>>> S = set([1, 2, 3])
>>> print(S)
>>> S.add(1)
>>> print(S)
>>> S.add(4)
>>> S.remove(3)
>>> print(S)
>>> S2 = S
>>> S2 is S
```

Q: Describe what happened? Anything unexpected?

Laboratory / Homework - The Birthday Paradox

The Birthday Paradox (or Problem) concerns the probability of two randomly selected people having anniversary at the same date (https://en.wikipedia.org/wiki/Birthday_problem).

Interesting enough, in a group of 70 people the probability is of 99.9% that two persons are born on the same day of the year, and a 50% chance is reached with only 23 people.

We want to check that out by simulating birthdays and checking the probabilities.

Laboratory / Homework - The Birthday Paradox

The ingredient of our simulation/algorithm are:

- A group of "**N**" persons
  - Notice that each person *represents* a birthday
- A **random generator** of birthdays
  - We are only concerned for same days in any given year
- A **checker** for "cosmic-twins"
  - For what matters, it is sufficient only a pair of "cosmic-twins"
- Do this process "**NX**" times to extract a probability measurement
  - We want to take the **average** of the occurrences of "twins"

Laboratory / Homework - The Birthday Paradox

```
from random import randint
```

```
def generate_birthdays():
    """
    Return a list of random birthdays
    """
    birthdays = []
    for _ in range(n_persons):
        birthdays.append(randint(1, 365))
    return birthdays
```

```
def check_birthdays(birthdays):
    """
    Return True if two birthdays collide
    """
    unique_birthdays = set(birthdays)
    return len(unique_birthdays) < len(birthdays)
```

```
n_hits = 0
for i in range(N_SIMULATIONS):
    birthdays = generate_birthdays()
    have_twins = check_birthdays(birthdays)
    n_hits += int(have_twins)
```

```
print('{:f}'.format(n_hits/N_SIMULATIONS))
```

```
N_SIMULATIONS = 100
```

```
n_persons = int(input("Give me the number of people: "))
```

Laboratory / Homework - The Birthday Paradox

Let's check if the algorithm works (once rebuilt) with the following group sizes (N):

- 10, 15, 20, 23, 25, 30
  - Do the results match your expectations?
    - https://en.wikipedia.org/wiki/Birthday_problem

Once that is done, we will modify it. The following should be improved:

- Also get the number of simulations (NX) from the user input;
- Use a *list-comprehension* in place of the *for* loop in *generate_birthdays()*