Python Tutorial

Peiyan (Vince) Gong
9/12/2018

What is Python?

1. Python is **Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
2. Python is **Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
3. Python is **Object-Oriented** − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
4. Python is a **Beginner's Language** − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Difference between interpreter and compiler:
https://www.programiz.com/article/difference-compiler-interpreter

(The following material refers to the following link:
https://cs231n.github.io/python-numpy-tutorial/#python-functions)

Basic Syntax:

Indentation:

Right way:

```
if True:
    print "True"
else:
    print "False"
```

Wrong way:

```
if True:
print "Answer"
print "True"
else:
print "Answer"
print "False"
```

Comments:

```
# This is a comment
```

Basic Data Type:

Numbers: (no x++ and x--!)

```
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"
```

Booleans:

```
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:

```
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"
```

(Some functions):

```
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;
                               # prints "he(ell)(ell)o"
print('  world '.strip())  # Strip leading and trailing whitespace; prints "world"
```

More is here: https://docs.python.org/3.5/library/stdtypes.html#string-methods

Containers:

Python includes several built-in container types: lists, dictionaries, sets, and tuples. Here I will emphasis the usage of **list**.

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])  # 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']"
```

More is here: https://docs.python.org/3.5/tutorial/datastructures.html#more-on-lists

(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]"
```

Careful: Python has different indexing with Matlab. In matlab, indexing with 2:4 means 2, 3, 4. In Python, indexing with 2:4 means 2, 3.

(Looping:)

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

Functions:

Python functions are defined using the **def** keyword.

```
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"
```

Here is a template for you to start with Python code:

```
def main():
   print ‘hello world!’

If __name__ == ‘__main__’
 main()
```

Numpy:

In order to use numpy library, you must add `import numpy as np` at the front of your code. (Just like other library)

```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
```

```
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

Array:

Numpy provides lots of ways to create special matrices:

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

(Slicing and Indexing:)

```
import numpy as np
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

```
# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

More about slicing and indexing here:
https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

(DataType:)

```
import numpy as np

x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

(Array Math:)

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
```

```
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

Careful: * in numpy is element-wise multiplication**.** You must use **dot** function to perform matrix multiplication:

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
```

```
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Careful: If you assign matrix A = matrix B, what you change on matrix B will have the same effect on matrix A!

```
>>> x = np.array([1, 2, 3])
>>> y = x
>>> z = np.copy(x)
```

Note that, when we modify x, y changes, but not z:

```
>>> x[0] = 10
>>> x[0] == y[0]
True
>>> x[0] == z[0]
False
```

Tips for Debugging and Programming:

1. Think carefully and plan things out before coding. (flow chart, pseudo code and simple graph)

ALGORITHM 1

1. Given n sensor readings from m sensors

2. Convert sensor readings into sensor motion $T_{i,k}^{i,k-1}$. Each transformation has the associated variance $(\sigma_{i,k}^{i,k-1})^2$

3. Set one of the lidar or nav sensors to be the base sensor B

4. Convert $R_{i,k}^{i,k-1}$ to angle-axis form $A_{i,k}^{i,k-1}$

For i = 1,...,m

5. Find a coarse estimate for R_i^B by using the Kabsch algorthim to solve $A_{B,k}^{B,k-1} = R_B^i A_{i,k}^{i,k-1}$ weighting the elements by $W_k = (max((\sigma_{i,k}^{i,k-1})^2) + max((\sigma_{B,k}^{B,k-1})^2))^{-0.5}$

6. Label all sensor readings over 3σ from the solution outliers

end

7. Find estimate for R_i^B by minimising

$$\sum_{i=1}^{m}\sum_{j=i}^{m}\sum_{k=2}^{n}\sqrt{Rerr_{ijk}^T\sigma^2 Rerr_{ijk}}$$

$$Rerr_{ijk} = (A_{j,k}^{j,k-1} - R_j^i A_{i,k}^{i,k-1})$$

$$\sigma^2 = (\sigma_{j,k}^{j,k-1})^2 + R_j^i(\sigma_{i,k}^{i,k-1})^2(R_j^i)^T$$

using the coarse estimate as an initial guess in the optimisation.

For i = 1,...,m

8. Find a coarse estimate for t_i^B by solving $t_i^B = (R_{i,k}^{i,k-1} - I)^{-1}(R_i^B t_{B,k}^{B,k-1} - t_{i,k}^{i,k-1})$ and weighting the elements by $W_k = (max((\sigma_{i,k}^{i,k-1})^2) + max((\sigma_{B,k}^{B,k-1})^2))^{-0.5}$

end

9. Find estimate for t_i^B by minimising

$$\sum_{i=1}^{m}\sum_{j=i}^{m}\sum_{k=2}^{n}\sqrt{Terr_{ijk}^T\sigma^2 Terr_{ijk}}$$

$$Terr_{ijk} = (R_{i,k}^{i,k-1} - I)t_i^j + t_{i,k}^{i,k-1} - R_i^j t_{j,k}^{j,k-1}$$

using the coarse estimate as an initial guess in the optimisation.

10. Bootstrap sensor readings and re-estimate R_i^B and t_i^B to give variance estimate $(\sigma_i^B)^2$

11. Find all sensors that have overlapping field of view

12. Use sensor specific metrics to estimate the transformation between sensors with overlapping field of view

13. Combine results to give final R_i^B, t_i^B and $(\sigma_i^B)^2$.

2. Use comment to track on what you are doing: Writing a function, what do you want the function do? Writing a equation, what do you want the equation to solve? Naming a parameter, what does this parameter represent?

3. Try to learn a good coding style: Make your code readable: Naming your variables and functions with reason. Use width, height, length, weight and etc instead of x,y,z,a,b,c. Good code should have very short and clear main function with just a few functions.

4. Devil is in the detail! Careful with typo, mis-sized matrix. Check everything and print everything.

5. Your code is not running or doing something weird, Just google it.

6. No one can get their code working in one shot, be patient.

7. **Start early! Start early! Start early!**