Introduction to Python
Robot Operating System

Autonomous Robotics Labs

Labs 01 (19.2./21.2. 2019)

Outline

- Intro to ARO labs
- Python
  - Basics
  - Built-in types
  - Flow-control statements
  - Functions
  - Classes
  - Packages
  - Example
- ROS
  - Basic concepts
  - Components & command line commands
  - Workspace

ARO Labs

- For details and contacts – please see the course web page
- Main assignment:
  - Develop a program for a real turtlebot

  - Explore a simple maze ($\sim 5 \times 5$ meters)
  - Find and retrieve an object
  - Bring it back to the start
- The first 7 labs should give you the basic knowledge needed to do this

Python

Python

- What is Python?
  - (very) high-level programming language
- Why Python?
  - loads of code for many problems, especially scientific & engineering (direct open-source/free Matlab competition)
  - ROS support
- Which Python?
  - Two major versions:
    - **2.7**
    - 3.x (currently 3.7)
  - Trap for young players – in Python 2.7:

```
print(7 / 2)   # 3
```

    - Integer divided by integer will result in an integer!

```
print(7 / 2.0)   # 3.5
```

  - Unfortunately, **ROS** supports only **Python 2.7**

How to install on your machine and where to code?

- Python is installed by default on many Linux distros, otherwise: https://www.python.org/downloads/
- Packages are installed via `pip` (e.g., `$ pip install numpy`)
- Optional package manager – Anaconda https://www.anaconda.com/distribution/
- Always remember that you need Python 2.7
- Coding environments:
  - PyCharm – preinstalled on faculty machines
  - VS Code – free, very lightweight IDE with support for many languages and community developed extensions; support for GIT
  - Spyder – free Python IDE with interactive (IPython) console – "free Matlab"
  - Jupyter notebook – IPython in your browser
  - Vim – for the hardcore Linux fans
  - many other environments exists...

Python programs

- Code organized into "script" files with ".py" extension
  - a script can be run either from an IDE or via command line:

    ```
    python my_script.py
    ```
- Larger code organized into packages and modules
  - packages -> basically directories containing script files (∼modules)

Import a module (or a package...):

```
import <module_name>
```

Import a components from a module:

```
from <module_name> import <component_or_class>
```

Import a module under a different name:

```
import <module_name> as <my_name>
```

And everything together:

```
from <module_name> import <component> as <my_component_name>
```

Import everything from a module:

```
from <module_name> import *  # <- NEVER DO THIS!!!
```

Basic Syntax

- The usual basic stuff:

```
2 + 3
a = 2
b = 3
c = a + b
```

...and so on.

- Again:

```
x = 2 / 3
print(x)  # 0
y = 2 / 3.
print(y)  # 0.666666667
```

  - Be careful about it!

- ```
  Python variables are not typed:
  a = 1
  a = "hello"  # no error!
  ```
  ```

Basic Syntax

- Comments:

```
# single line comment

"""Multiline
comment
"""

'''
Single quotes work as well
'''
```

(although, the multiline comment is just a multiline string that is not stored or printed out)

- Output:

```
print "Hello"  # works in 2.7 only
# recommended:
print("Hello")  # works in both 2.7 & 3.x
```

Strings

- Can be specified using either single or double quotes (always matching)
- Adding variables:

```
temp = 37
humid = 70
print("The temperature is {} degrees \
        with {}% humidity.".format(temp, humid))
# Prints:  The temperature is 37 degrees
# with 70% humidity.
```

- There are many was to format a string, search for “python string formatting”

Lists I

- "arrays" or as Python calls them – "lists":

```
l = [1, 2, 3, 4, 5]
print(l)  # [1, 2, 3, 4, 5]
print(l[0])  # 1

l.append(6)  # append 6 at the end of the list
a = l.pop()  # simple "stack" - LIFO
print(a)  # 6

l[2] = 11  # assign 11 to the 3rd element
print(l)  # [1, 2, 11, 4, 5]
```

- multidimensional or "nested" list:

```
l2D = [[1, 2, 3], [4, 5, 6]]
print(l2D[1][2])  # 6
```

- lists from ranges and how to convert other objects to lists:

```
lrange = list(range(10))  # 2.7 does not need list()
```

Lists II

- The blessing & the curse – Python variables are not typed:

```
mixedList = ["a", 2, myObject]
```

- Length of an array (or any other iterable):

```
print(len(lrange))  # 10
```

- Check if a list contains a value:

```
print(3 in lrange)  # True
```

- Lists are good (especially because of list comprehensions, shown later) but for more complex operations use **numpy arrays** (also shown later)

Sets

- "lists" with unique elements – i.e. math-like sets

```python
mySet = set(["a", "b"])
mySet = {"a", "b"}

print(mySet)  # {'a', 'b'}
print("a" in mySet)  # True

mySet.add("c")
print(mySet)  # {'a', 'b', 'c'}
mySet.add("a")
print(mySet)  # {'a', 'b', 'c'}
```

- Usual mathematical set operations are possible (union, intersection,...)
- Immutable version:

```python
myFrozenSet = frozenset(mySet)
```

Tuples

- ordered immutable lists
- Why?
  - functions can return multiple values (more on that later)
  - slightly faster
  - the usual immutable stuff (e.g., comparison of two variables)

```
myTuple = tuple([1, 2])
myTuple = (1, 2)
singleTuple = (1, )  # note the comma!
```

- Indexing works the same way as it does with lists:

```
print(myTuple[0])  # 1
```

- Immutable?

```
myTuple[0] = 5  # error!
```

Dictionaries

- Unordered "hash tables"

```
d = {"key1": "value1", "key2": 2}

print(d["key1"])  # "value1"
print(d["key2"])  # 2

d["newKey"] = "newValue"
print(d)  # {'key2': 2, 'key1': 'value1', 'newKey': 'newValue'}
```

- Ordered dictionary:

```
from collections import OrderedDict
orderedDictionary = OrderedDict()
```

- remembers order of insertion

collections & more

- The Python package *collections* contains more useful classes
- Queue (FIFO):

```
from Queue import Queue
q = Queue()
q.put(1)
q.get()
```

– this is in contrast to the previously shown "stack-like" (LIFO) behavior of normal lists

- Double sided list (faster left-append or prepend, if you will)

```
from Queue import deque
dq = deque()
dq.append(1)
dq.appendleft(2)

print(dq)  # deque([2, 1])
dq.popleft()  # 2
dq.pop()  # 1
```

Conditions

```
if <condtion>:
        pass   # "pass" does nothing, not even an error
elif <another_condtion>:
        pass
else:
        pass
```

- Mind the "tab" space – important part of code structure (unfortunately)
  - consecutive lines with the same amount of whitespace before them exist in the same scope
  - use **spaces** instead of tabs (most good Python IDEs insert spaces when tab key is pressed)
- No "select-case" statements – can only be implemented with if-elif
- Ternary operator (i.e. in-line condition):

```
w = 5
v = "a" if w > 5 else "b"
print(v)  # b
```

While Loops

- Loop that will continue until the condition is met:

```
while <condition>:
        <do_stuff>
```

- Example:

```
a = 1
while a < 10:
        a += 2
        print(a)  # 3, 5, 7, 9, 11
```

- More complex example:

```
a = 1
while a < 10:
        a += 2
        if a == 5:
                continue  # skips the rest of the current loop
        print(a)
        if a > 8:
                break  # breaks out of the loop
# Prints: 3, 7, 9 (5 is skipped and breaks before 11)
```

For loops

- Classic loop iterating through a sequence of numbers:

```
for i in range(10):
        print(i)
# Prints numbers from 0 to 9 (10 is not included!)
```

- For loop actually iterates ("goes through") any ***iterable***:

```
for elem in [4, 6, 8, 12]:
        print(elem)
# Prints 4 6 8 12
```

```
d = {"a": 1, "b": 2, "c": 3}
for key, value in d.items():
        print(key, value)
# Prints: ('a', 1) ('c', 3) ('b', 2)
```

- *enumerate* keyword can be used to "attach" ordering number to the loop variable:

```
for i, (key, value) in enumerate(d.items()):
        print(i, key, value)
# Prints: (0, 'a', 1) (1, 'c', 3) (2, 'b', 2)
# note that "items()" returns a tuple
```

Loops - list comprehensions

- Special Python construct
- A more elegant and sometimes faster way of creating lists

```
expList = [x**2 for x in range(10)]
print(expList)
# Prints: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Can contain conditions:

```
print([x**2 if x < 5 else 2 * x for x in range(10)])
# Prints: [0, 1, 4, 9, 16, 10, 12, 14, 16, 18]
```

- You can get very crazy with list comprehensions, just be careful

```
simple2DList = [[a + b for a in range(5)] for b in range(10)]
```

```
complex2DList = [[a + b for a in range(5)] if b % 2 == 0
else [a * b for a in range(3)] for b in range(10)]
```

...and so on.

Functions I

- Specified with keyword "***def***"
- No difference between functions and procedures

```
def <function_name>(<args>):
        <do_stuff>
```

- May or may not return a value

```
def example(msg, randomNumbers, report=True):
        a = []
        for num in randomNumbers:
                a.append(msg + str(num))
                if report:
                        print(a[-1])
        return a

q = example("Hello", [3, 5, 4])
```

Functions II

- If no value is explicitly returned, the function returns a special "**None**" type:

```
def void():
    print("I shan't return anything!")

ret = void()
# outputs: "I shan't return anything!"
print(ret)   # None
print(ret is None)  # check if None was returned
```

- As promised: tuple return value:

```
def tupler(value):
    oneLower, oneHigher = value - 1, value + 1
    return oneLower, oneHigher

print(tupler(3))  # (2, 4)
```

Functions III – arguments

```
def fun(alpha, beta="value", *args, **kwargs):
    print("Alpha: ", alpha, " Beta: ", beta)
    for argument in args:
        print(argument)
    for key, value in kwargs.items():
            print(key, ": ", value)

fun("a")
fun("a", "other")
fun("a", "other", 1, 2, 3)
fun("a", "other", 1, 2, 3, custom=4, myoption="something")
fun("a", custom=4, myoption="something")
l = [1, 2, 3]
fun("a", "other", *l)
d = {"beta": "myvalue", "custom": 4, "myoption": "something"}
fun("a", **d)
```

- In function call:
  - "*" unpacks a list into the function arguments
  - "**" unpacks a dictionary into the function arguments
- In function definition:
  - "*" consumes any number of "simple" arguments
  - "**" consumes any number of keyword arguments

Classes

- Specified with a keyword "***class***"

```
class MyClass():

        def __init__(self, value=5):
                self.value = value

        def do(self, num):
                print(self.value + num)

mc = MyClass(3)
mc.do(4)   # 7
```

- Checkout magic functions to do some magic with classes: https://rszalski.github.io/magicmethods/

```
      def __getitem__(self, value):
            return "You wanted to return somehting\
                   at index {}".format(value)

mc = MyClass()
print(mc[7])  # 'You wanted to return somehting at index 7'
```

More packages

- `os` Functions related to the OS, e.g., *os.path* to manipulate paths
- `sys` System functions (e.g. PATH variable)
- `numpy` Huge set of math related functions and arrays
- `scipy` Whatever was not in numpy
- `matplotlib` Set of plotting functions
- `__future__` Set of Python 2.7→3.x compatibility modules
  - `print_function` Enforces the use of print() as a function
  - `division` enables "true division" (instead of integer division)

```
from __future__ import print_function, division
```

Numpy

- Extensive Python-enhancing library

```
import numpy as np
```

  - The convention is to use alias "np" – it gets used a lot so you want it to be short
- Perhaps most important contribution: arrays

```
arr = np.zeros((4, 6), dtype=np.int16)
arr[:, 2] = 7  # every row of the 2nd column
arr[1, 1:4] += 3  # first row in column 1 to 3
arr *= 2
print(arr.shape)  # (4, 6)
```

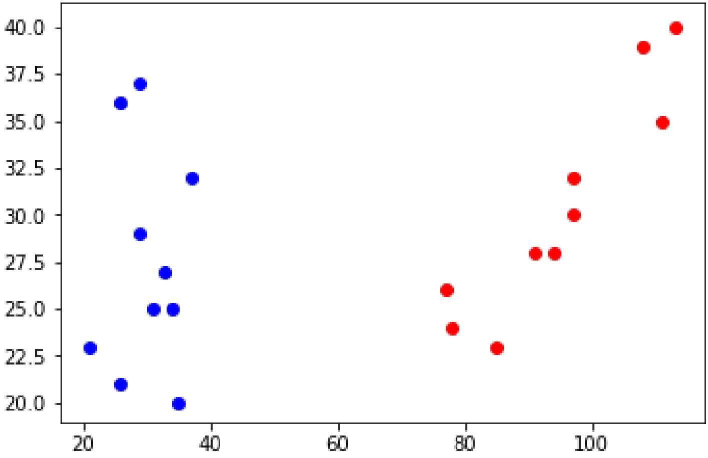
  - Extensive indexing and array manipulation capabilities
- Contains also **matrix** class for matrix and vector manipulation. In most cases, however, arrays are the more suitable approach.

Example

```python
import numpy as np # import the numpy package
from matplotlib import pyplot as plt # plotting library

points = np.random.randint(20, 40, (2, 10)) # random 2D points
# augment points with ones -> homogeneous coordinates
points = np.vstack((points, np.ones(points.shape[1])))

# create translation
tf_translate = np.matrix([[1, 0, 10], [0, 1, 3], [0, 0, 1]])
# create skew
tf_skew = np.matrix([[1, 2, 0], [0, 1, 0], [0, 0, 1]])

# transform points
tf_points = np.array(tf_translate * tf_skew * points)
"""Alternative without np.matrix:
tf_points = tf_translate.dot(tf_skew.dot(points))
"""

# plot the original and transformed points
plt.scatter(points[0, :], points[1, :], c="b")
plt.scatter(tf_points[0, :], tf_points[1, :], c="r")
```

Example

- (Possible) result:



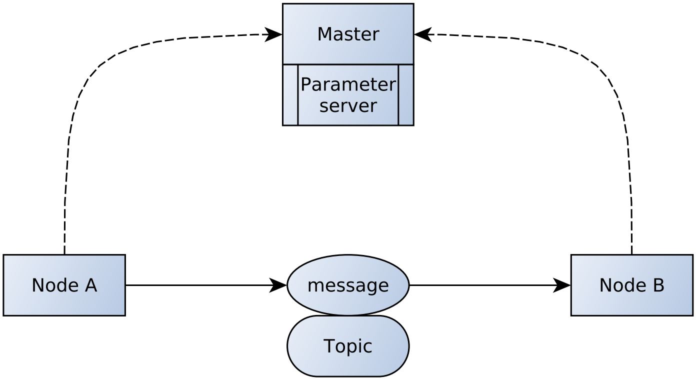
ROS

Very Fast & Furious ROS overview

- What is ROS?
  - Robot Operating System
  - asynchronous data processing (but can also operate in synchronous mode)
  - distributed system (but has a central "node")
  - contains a lot of "stuff" useful for developing SW for robotic applications:
    various tools (*packages*) & libraries for many robotics-related problems, SW management tools, visualization & debugging tools

ROS components

The simplest ROS topology:



ROS Master

- Communication “server” (ROS actually uses P2P model): **mediates communication between nodes**
  - every new node registers with the master (address where other nodes can reach it)
  - tracks topic and service publishers and subscribers
  - data is then sent directly between nodes
- Provides parameter server
- Always needs to be executed before doing anything else with ROS
  - `$ roscore`
  - launch files start master if not running already (I’ll explain later...)
  - run it & forget about it (until you get to more advanced stuff)
    - reasons for restarting: new logging session, cleaning up (crashed nodes – *`$ rosnode cleanup`*, renew parameter server)
    - cost of restarting: no new connections can be established -> whole system restart likely required
- Can be run on another machine on the network
  - `$ echo $ROS_MASTER_URI`
    `http://localhost:11311`
  - `$ export ROS_MASTER_URI=http://<other_machine>:11311/`
- Starts `/rosout` node – mostly for debugging

ROS Node

- Basic building block of ROS
- Executable programs and scripts (Python)
  - write a script
  - make it executable: `$ chmod u+x <filename>.py` or `$ chmod +700 <filename>.py`
  - run it: `$ rosrun <package_name> <node_name>.py`
    - simply executes an executable program or script
- *A node* is an **instance of** a ROS program
  - multiple instances of the same program can run simultaneously (with different names)
  - names separated into namespaces (/)
- Nodes can do anything you want them to (or anything you can program them to do)
- Communicate with other nodes via topics and services
  - can be all on one machine or distributed across the Universe, as long as they can all reach the *master* and each other
- Each node can be written in any language with ROS support: C++, Python, MATLAB, Java, Lisp

ROS Node: `console commands`

`$ rosnode`

`list`	**lists** currently active **nodes**; hint: `<command> | grep <expression>` outputs only lines containing the expression and highlights the occurrences
`info <node_name>`	shows **info about** a specific node – **topics** where the node **publishes** and to which it is **subscribed** to and **services** & node **address**
`ping <node_name>`	tests **node reachability** and response time
`machine [machine_uri]`	**lists machines** with nodes connected to the master or **nodes** running **on a** specific **machine**
`kill <node_name>`	does what it says on the cover...

Help will always be given to those who ask for it:

- `$ rosnode help`
- `$ rosnode <command> -h`

Or in general:

- `$ ros<whatever> help`
- `$ ros<whatever> <some_sub_command> -h`

And use TAB key!

- Trivia: Every time someone does not use command completion a cute bunny eats a fluffy unicorn! And bunnies have a lethal allergy to unicorn fur!

ROS Topic

- Communication channels used by the nodes to send and share information
- Publisher & Subscriber model
  - every node can publish or subscribe/listen to a topic
- Each topic has a specific data type that can be sent over it

ROS Topic: `console commands`

`$ rostopic`

`list`	creates tear in space-time fabric…nope, just **lists existing topics**; existing topic = any topic that was registered with the master, i.e. existing does not mean active (useful to know when debugging); use grep…
`info <topic_name>`	yup, prints **info about** a specific topic: **nodes publishing** in the topic, **subscribed** nodes and **type of message** that can be transferred via the topic (data type)
`hz <topic_name>`	shows **publishing rate** of a topic (better than echo if you just want to see whether something is being published over a topic)
`echo <topic_name>`	**writes out messages** transmitted over a topic (useful for debugging of topics with low rate and small messages); specific parts of the message can be printed by appending "/<msg_part>/…" `-noarr` flag will suppress printing of arrays
`type <topic_name>`	prints the **type of the messages** transmitted via the topic
`bw <topic_name>`	**bandwidth** used by the topic, i.e. the amount of data transmitted over it per second (on average) – useful to check when sending a lot of data
`pub <topic_name> <message_type> <msg>`	can be used to **publish a message** over a topic when debugging – obviously, only usable for topics with simple messages
`find <message_type>`	**lists** all **topics that use the** specified **message type**

ROS Message

- Data structures used to send data over topics
  - simple: bool, int<N>, uint<N>, float<N>, string, time, duration
  - complex: composed of simple types, can contain other message types and a header
- Message header

  `seq` sequence number – unique ever-increasing ID
  `stamp` message timestamp – epoch seconds & nanoseconds
  `frame_id` frame ID – frame associated with the message

  - `$ rostopic echo /<some_interesting_topic>/header` – will display just the headers of the messages
- Messages are defined in "message files"

ROS Message: `console commands`

`$ rosmsg`

show <message_name>	**shows** message **fields** (msg definition file)
`list`	**lists** all available **message types**
package <package_name>	**lists** all **message types** defineadditional args to provide package author, description, ...d **in a** specific **package**
`packages`	**lists** all **packages containing** (definitions of) any **messages**

Workspace

Workspace

- Collection of folders with related ROS files
- Source files, definitions, configuration files, scripts, and other files are organized into packages
- Compilation done **only** via the ROS build system

ROS Build system

- **catkin**
- a.k.a. *catkin command line tools*
  https://catkin-tools.readthedocs.io/en/latest/cheat_sheet.html
- Extension of CMake – can build libraries, executables,... (C++)
  - collection of CMake macros and Python scripts
- Auto-generates message/service/action related functions based on their definitions

`init`	initializes a workspace in the current folder
`config`	show current WS configuration (additional args to change the current config)
`create pkg <package_name>`	creates a new package (in the current folder); additional args to provide package dependencies, author, description, ...
`build [package_name]`	builds the current WS/package
`clean [package_name]`	cleans catkin products (build, devel, logs)

- Building a WS with catkin creates these folders in the WS:

  `build` build targets
  `devel` (as in "development") – contains setup script
  `logs` build logs

ROS Packages

- ROS files are organized into packages
- Structure of a package:

<some_package>

[src]/package_name/ source code – scripts; normal "Pythonic" code structure

[scripts] usually (non-Python/non-C++) scripts or (standalone) executables

[launch] launch files

[config] configuration files, yaml param files for param server

[include] additional libraries; include headers for C++

[msg] message definitions

[srv] service definitions

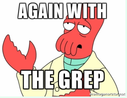
[action] action definitions

CMakeLists.txt CMake config file (used by catkin)

package.xml package manifest – catkin/ROS package config filelogs build logs

ROS Packages: `console commands`

$ rospack

list	**lists** all currently available **packages**	
find <message_name>	prints **path** to a specific **package**	

$ roscd <package_name> – *cd* into a package

$ rosls <package_name> – *ls* a package directory content

$ rosed <package_name>/<some_file> – launch a text editor and open the specified file in it (a quick way to adjust small details in a file while debugging)

Creating a workspace

- **Create** folder and *cd* into it
  `$ mkdir example_ws && cd example_ws`
- Create **src** folder
  `$ mkdir src`
- **Init** the workspace
  `$ catkin init`
- **Build** the WS (builds just the catkin tools)
  `$ catkin build`
- Look at it (just to make you feel happy)
  `$ ll` or `$ ls -la` (if the first command does not work
- Go into the src folder
  `$ cd src`

Creating a package

- Create a package
  `$ catkin create pkg incredible_package --catkin-deps rospy`
- CD into the package
  `$ cd incredible_package`
- Check and modify the manifest
  `$ vim package.xml` (or just use GUI based editor)
- Check the CMakeLists.txt (just look at it for now)
- Create a src folder (if it does not exist)
  `$ mkdir src/`

Creating a node

- Fire up your favorite editor and create publisher.*py*:

```
#!/usr/bin/env python2
import rospy
from std_msgs.msg import Float32
from numpy.random import rand

if __name__ == '__main__':
    rospy.init_node('publisher')
    rate = rospy.Rate(2)
    publisher = rospy.Publisher('random',
                    Float32, queue_size=10)
    while not rospy.is_shutdown():
        publisher.publish(rand())
        rate.sleep()
```

- Make executable
  `chmod u+x publisher.py`
- Build & source
  `$ catkin build`
  `$ source ~/example_ws/devel/setup.bash`

Creating another node

- *listener.py*

```python
#!/usr/bin/env python2
import rospy
from std_msgs.msg import Float32

def callback(msg):
    print('Received a message: {}'.format(msg))
    # rospy.loginfo('Received a message:\
    # {}'.format(msg))

if __name__ == '__main__':
    rospy.init_node('listener')
    publisher = rospy.Subscriber('random',
                    Float32, callback)
    rospy.spin()
```

You first ROS package

- Run the nodes and observe the beauty of messages being transmitted:

```
$ roscore
```

```
$ rosrun my_package publisher.py
```

```
$ rosrun my_package listener.py
Received a message: data: 0.312089651823
Received a message: data: 0.984019577503
Received a message: data: 0.142692148685
Received a message: data: 0.230828240514
Received a message: data: 0.27526524663
```

Thank you for your attention