RT Embedded
Linux Solutions

Real Time Group

Python 3

RT Embedded Linux Solutions

Glossary (1/2)

Gloassary (2/2)

RT Embedded Linux Solutions

Sockets	162
Serial	175
Sqlite	187
Multithreading	200
tkinter	208

Introduction

RT Embedded
Linux Solutions

Introduction to Python

Python is a highly popular scripting language used for a wide variety of applications, both programming and automation.

Here are some of its features and benefits:

- High-level programming language (supports OOP)
- Open source and community driven
- Supported by a wide variety of Operating systems, platforms and distributions (including Linux)
- Portable code, easy to maintain and develop.
- Short and effective source files (about 1:3 to 1:5 code ratio compared to other popular languages), yet very intuitive.
- Dynamic typed, Source can be “compiled” to run instantly
- Similar to c/c++, perl, tcl, ruby

Who uses Python ?

Because Python has been around for over two decades, it has a very large crowd and a very active developer community.

This indicates a very robust and stable programming / scripting language. This is the reason why python is being applied in real revenue-generating products by real companies.

The exact number of users is hard to determine science there is no license registration for python’s installation, and many Operating Systems include python by default, such as Mac OS and the vast majority of Linux Distributions.

However, it is estimated over a million users, based on various statistics (according to download rates and developer surveys).

Who uses Python ?(2)

Maya, a powerful integrated 3D modeling and animation system, provides a Python scripting API.

Industrial Light & Magic, Pixar, and others use Python in the production of animated movies.

NASA, Los Alamos, Fermilab, JPL, and others use Python for scientific programming tasks.

The NSA uses Python for cryptography and intelligence analysis.

Python is optimal for testing Kernel Modules and Drivers for embedded systems and general purpose OS (e.g linux).

And so on …

What can I do with Python ?

In addition to being a well-designed programming language, Python is useful for accomplishing real-world tasks (the sorts of things developers do on a day to day basis).

As mentioned before, It's commonly used in a variety of domains, as a tool for scripting other components and implementing standalone programs.

In fact, as a general-purpose language, Python's roles are virtually unlimited: you can use it for everything from website development and gaming to robotics and spacecraft control.

What can I do with Python ? (2)

System Programming / Scripting

The fact it comes with most Linux distribution (and Mac operating systems) makes it ideal for automating the OS. You can write portable utilities and admin-tools with ease.
Python can manage files and directory trees, launch other programs, do parallel processing (processes & threads) etc.

Gui Support

Python has a standard API for gui apps, called Tk GUI API or Tkinter. GUI runs unchanged on all major OS (Windows, Linux, Unix, MacOS).

An alternative GUI API can be used, called wxPython. It is newer and uses C++ libs, and has the same features as Tkinter.

What can I do with Python ? (3)

RT Embedded Linux Solutions

Gui Support with high level programming/scripting

With the proper library you can integrate with other common GUIs. Part of the next examples are built-in in python:

- PythonCard and Dabo (uses wxPython and Tkinter API)
- PyQt (uses QT)
- PyGtk (uses GTK)
- PyWin (uses windows MFC)
- IronPython (uses .NET)
- Jython and jPype (uses Java Swing)

Python provides Active Directory modules for automation and maintenance, an attractive benefit for system administrators.

What can I do with Python ? (4)

RT Embedded
Linux Solutions

Internet and Network Programming / Scripting

Python comes with standard Internet modules that allow Python programs to perform a wide variety of networking tasks, both Peer-To-Peer and client-server modes:

- Python Scripts can use sockets (client / server / PTP)
- Transfer file using FTP, SFTP, NFS etc..
- Parse, generate and analyse XML / HTML content, and also send, receive and compose emails.
- In addition there is a large collection of third-party tools and even full development framework packages for python (Django, TurboGear, web2py etc..). Many of them include a complete and enterprise-level web development solutions.

What can I do with Python ? (5)

RT Embedded Linux Solutions

Database support

Python interfaces with all common DB systems. Such as MySql, Microsoft SQL Server, Sybase, Oracle SQL Server, Informix, ODBC, SQLite and more ..

Python has also a portable database API for accessing SQL DB systems from python scripts. You can switch DB engine as you please and suffer a minimal change of code for the new DB.

Even more with python:

Easier development of games with the pygame package.

Serial port communication support with pyserial extension.

Image processing with PIL, PyOpenGL, Blender, Maya ..

We won’t be able to cover all of its domains, but you get the concept ...

Installation

The tools we will use

Python 3 interpreter - will allow us to run Python 3 code (.py files) and to run the python 3 interactive interpreter

pip - Python package installation tool. will allow us to install new modules for python 3

PyCharm - Our IDE (Integreated Development Editor) of choice. Will make it easier to develop and test python code.

Note: If you already have experience with code editors you can choose whichever editor or IDE you're comfortable with. Alternatively you could install/work-with any of the following programs:

Eclipse - A commonly used free IDE with the ability to develop in Java Python C C++ and more. May require plugins
Notepad++ - a simple text editor for Windows with colorization and simple auto-complete.
Geany - Text editor for Linux with colorization, simple auto-complete and the ability to directly run python code

Installation on Linux

Installing Python on Linux (Ubuntu 16.04)

Python 3 should already be installed on most linux distributions, but pip and tkinter might not be installed.

To install python3, pip and tkinter open a terminal (Ctrl+Alt+T) and execute the following command:

```
sudo pip install python3 python3-pip python3-tk
```

(press enter, to install and press enter when asked if you wish to continue)

Try executing the following commands:

python3 --version

- will print the version of python you have installed

pip3 --version

- will print the version of pip for python 3 you have installed

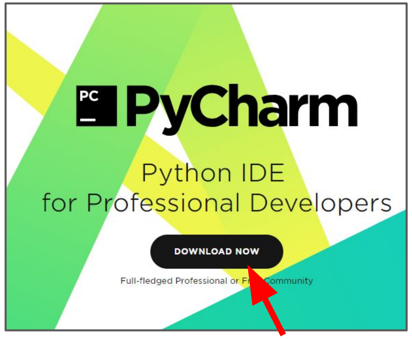
python3 -c “import tkinter”

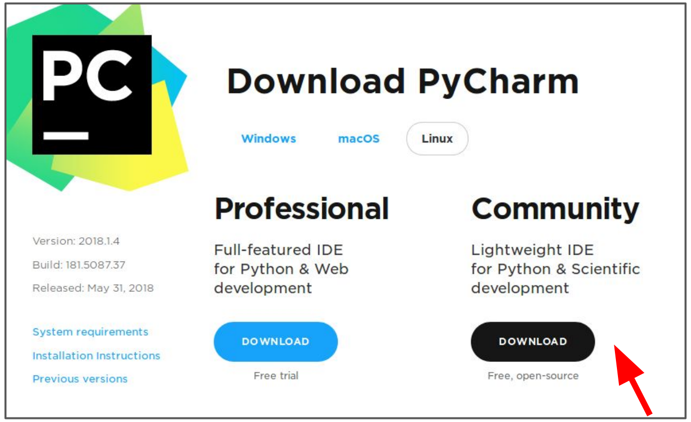
- should print nothing. Otherwise will print an error if tkinter for python 3 is **not** installed

Downloading the PyCharm IDE on Linux

RT Embedded Linux Solutions

Go to: https://www.jetbrains.com/pycharm/
Select “Download”. Then, select the download below the “**Community Edition**” column. save the .tar.gz file.
Extract the tar.gz file and copy the folder to where you want to install it.


PyCharm
Python IDE
for Professional Developers
DOWNLOAD NOW
Full-fledged Professional or Free Community

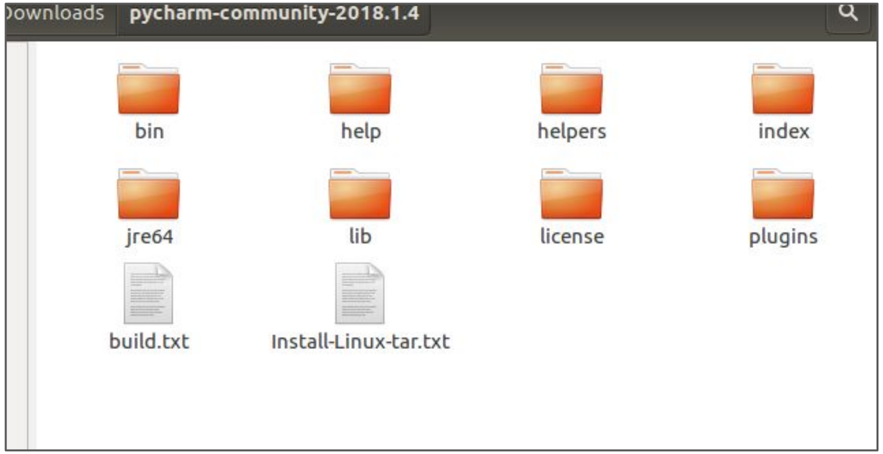


Installing the Pycharm IDE on Linux

RT Embedded Linux Solutions

Open the extracted folder, go to the bin folder. right click and open a terminal in that folder. execute the following command:
./pycharm.sh

Continue following from page 24 “**Configuring PyCharm**”.



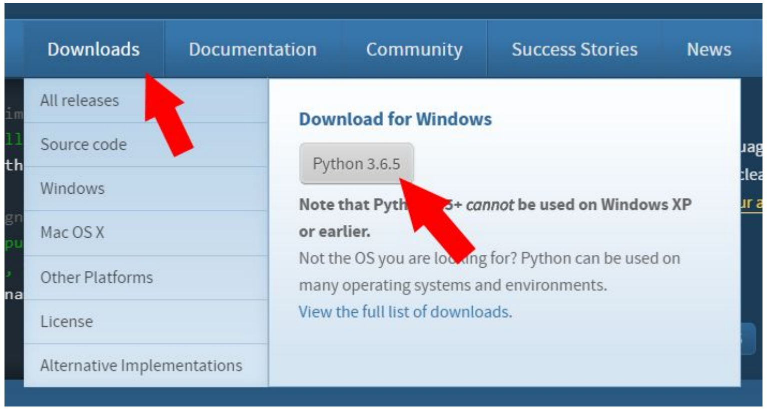
Installation on Windows

Installing Python on Windows

RT Embedded Linux Solutions

Windows

You can download an installer from **python.org**. Select the **Downloads** tab and then select the button showing the python version:



Note that Pyth 5+ *cannot* be used on Windows XP or earlier.

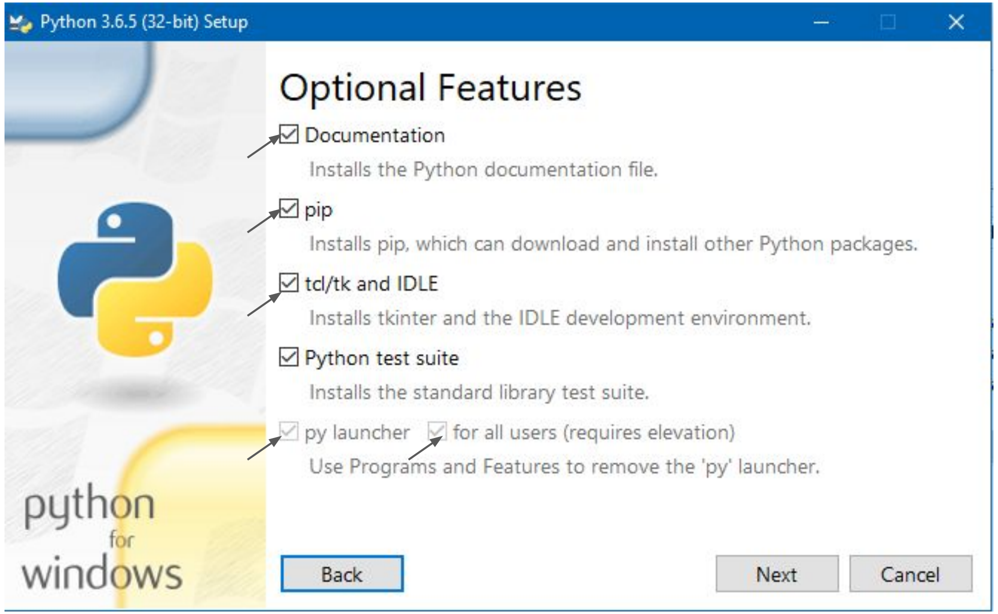
Not the OS you are lo ing for? Python can be used on many operating systems and environments.

View the full list of downloads.

installing pip and tkinter on windows

RT Embedded Linux Solutions

While installing Python 3 make sure that "**Documentation**", "**pip**", "**tcl/tk and IDLE**", "**py launcher**" and "**for all users**" are marked before pressing "**Next**".



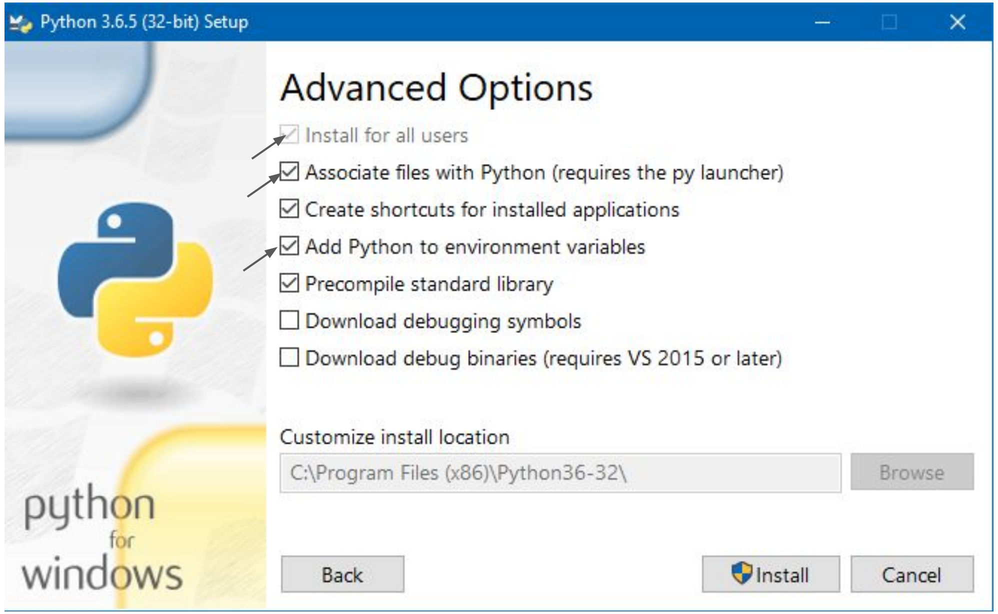
Python 3.6.5 (32-bit) Setup

Optional Features

- ☑ Documentation
  Installs the Python documentation file.
- ☑ pip
  Installs pip, which can download and install other Python packages.
- ☑ tcl/tk and IDLE
  Installs tkinter and the IDLE development environment.
- ☑ Python test suite
  Installs the standard library test suite.
- ☑ py launcher ☑ for all users (requires elevation)
  Use Programs and Features to remove the 'py' launcher.

Instaling Python3 on windows: Advanced Options

Make sure “**Install for all users**” “**associate files with Python…**” and “**Add Python to enviroment variables**” are marked before pressing **Install**.



Python 3.6.5 (32-bit) Setup

Advanced Options

- [✓] Install for all users
- [✓] Associate files with Python (requires the py launcher)
- [✓] Create shortcuts for installed applications
- [✓] Add Python to environment variables
- [✓] Precompile standard library
- [ ] Download debugging symbols
- [ ] Download debug binaries (requires VS 2015 or later)

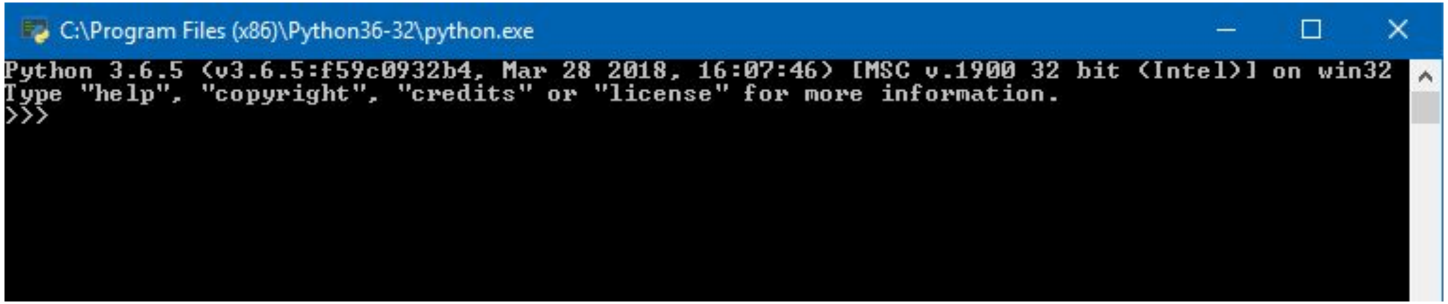
Customize install location

C:\Program Files (x86)\Python36-32\

Testing Python installation on Windows

RT Embedded Linux Solutions

press Ctrl+R to open the run menu, type “python” and press enter (if that doesn’t work try typing “py” instead)
You should see a terminal window running python similar to the one shown below:



C:\Program Files (x86)\Python36-32\python.exe

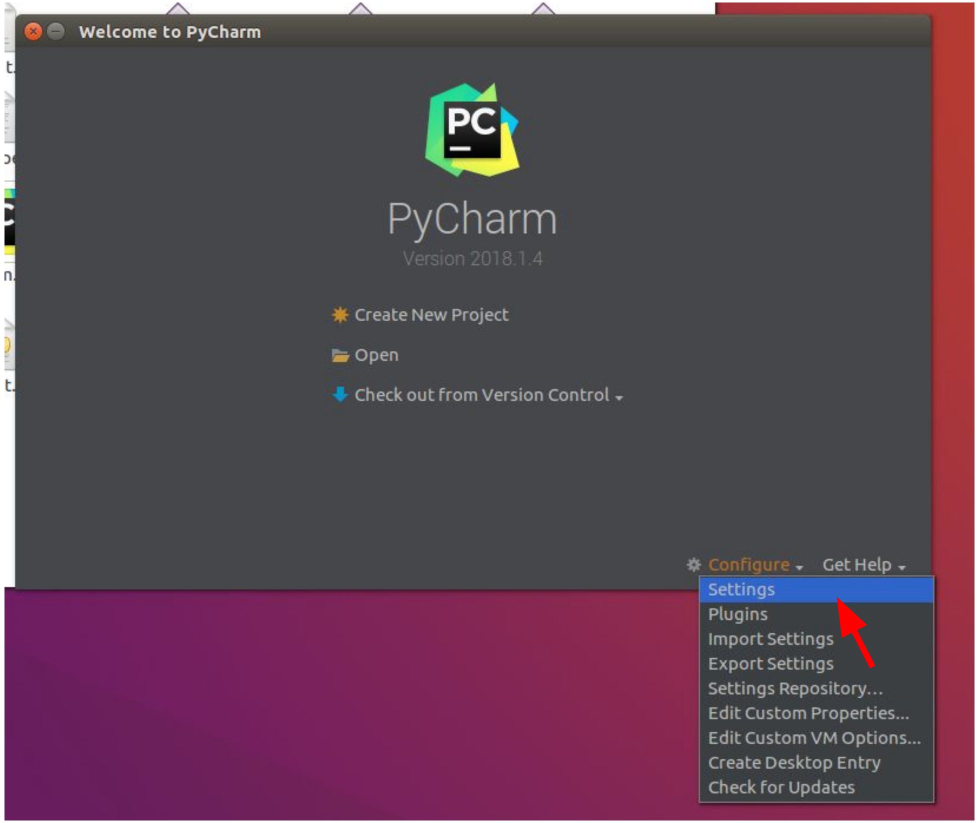
```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Configuring PyCharm

Adding Python Interpreter (1/3)

RT Embedded Linux Solutions

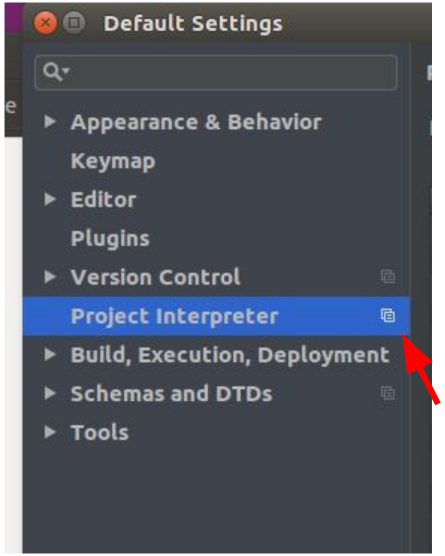
Open Pycharm and click “configure” on the bottom and click “settings”.

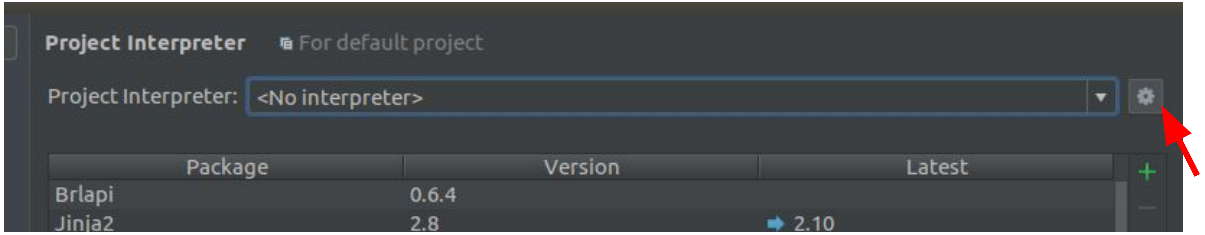


RT Embedded Linux Solutions

Adding Python Interpreter (2/3)

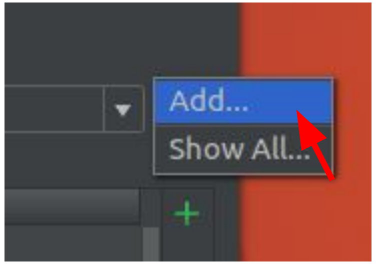
Choose "Project Interpreter" from left pane. Click on cog wheel icon from right pane. choose "add…".




Project Interpreter   For default project

Project Interpreter: <No interpreter>

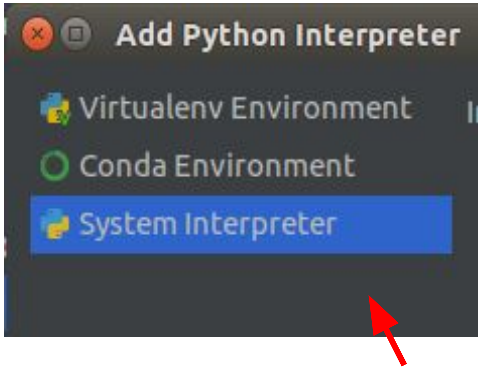
Package	Version	Latest
Brlapi	0.6.4	
Jinja2	2.8	➡ 2.10

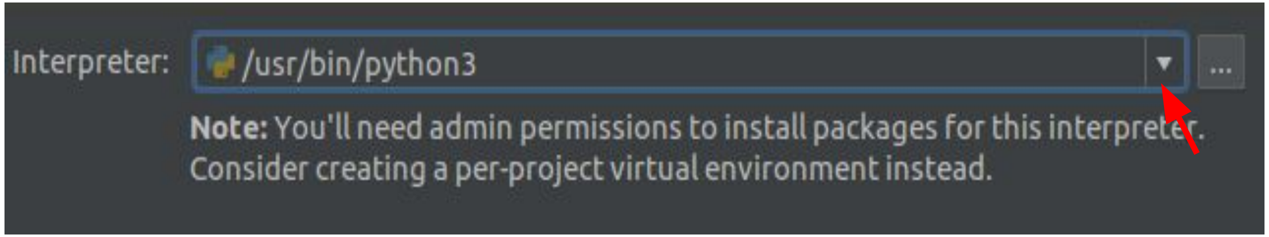


Adding Python Interpreter (3/3)

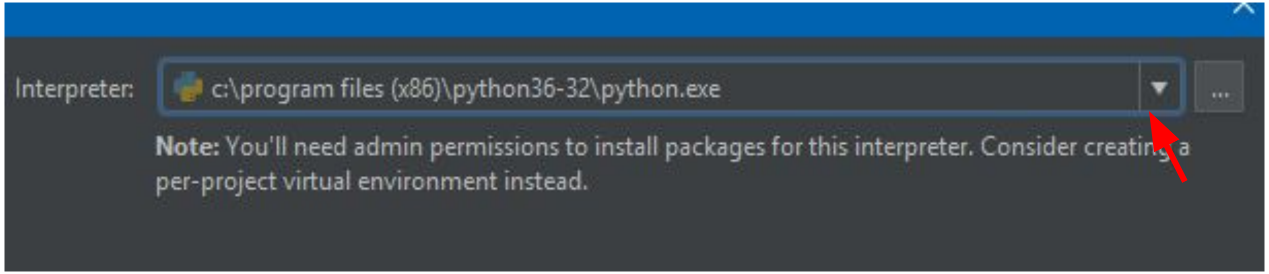
RT Embedded Linux Solutions

Select “System Interpreter” from the left pane. Select your interpreter from the right dropdown menu (if not already selected)
(On windows its “c:\program file (x86)\python36-32\python.exe”)
(On Linux its “/usr/bin/python3”)




Interpreter: /usr/bin/python3

Note: You'll need admin permissions to install packages for this interpreter. Consider creating a per-project virtual environment instead.


Interpreter: c:\program files (x86)\python36-32\python.exe

Note: You'll need admin permissions to install packages for this interpreter. Consider creating a per-project virtual environment instead.

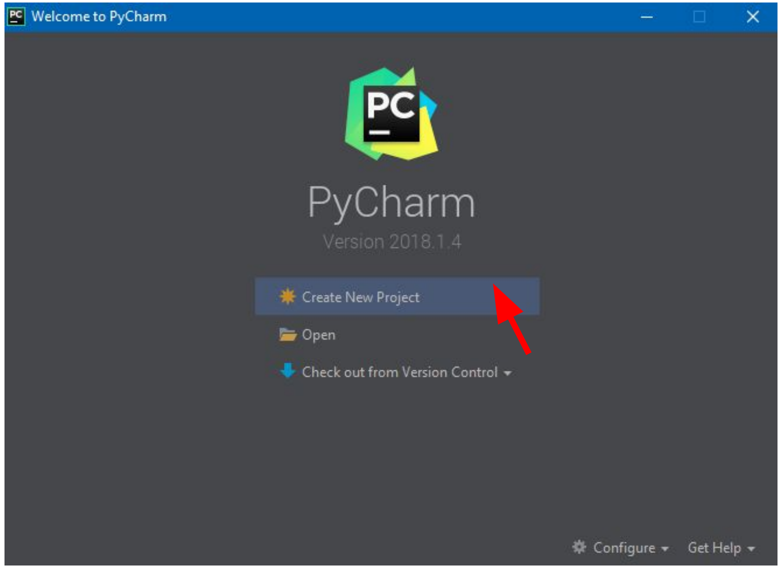
RT Embedded
Linux Solutions

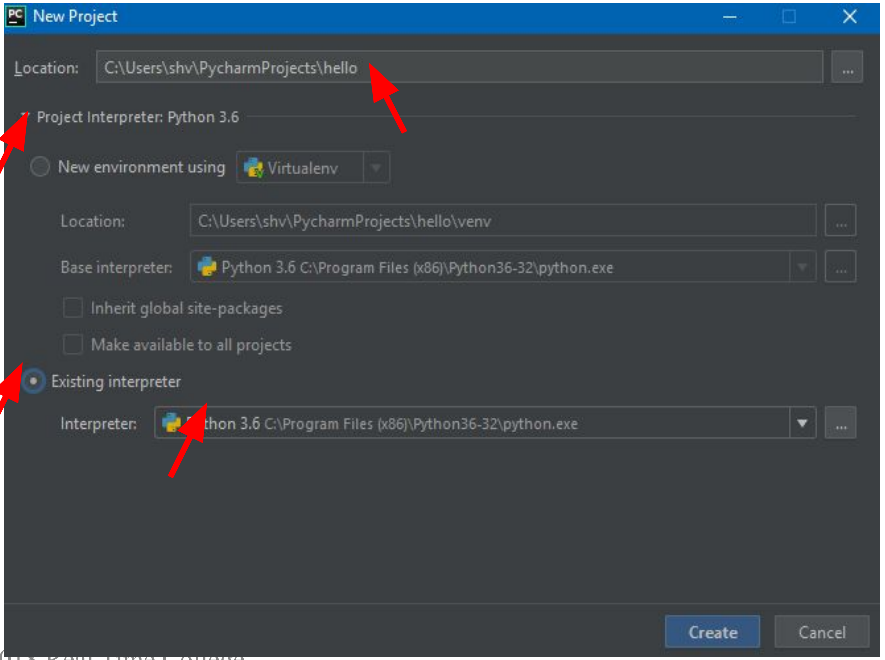
Hello World!

Starting a new Project

RT Embedded Linux Solutions

From the Welcome windows click “Create New Project”.
In the new project window, type the name of the project (“hello”), select “project interpreter”, make sure “**existing interpreter**” is selected and that the interpreter we configured earlier is selected.





New Project

Location: C:\Users\shv\PycharmProjects\hello

Project Interpreter: Python 3.6

New environment using Virtualenv

Location: C:\Users\shv\PycharmProjects\hello\venv

Base interpreter: Python 3.6 C:\Program Files (x86)\Python36-32\python.exe

Inherit global site-packages

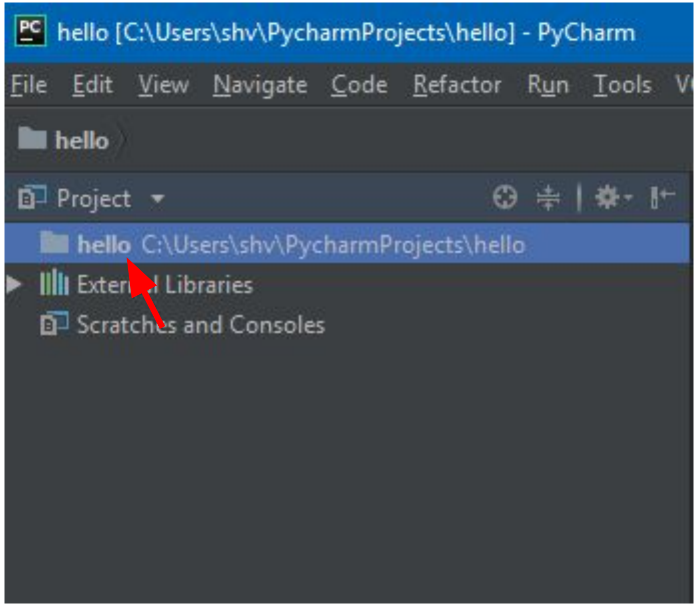
Make available to all projects

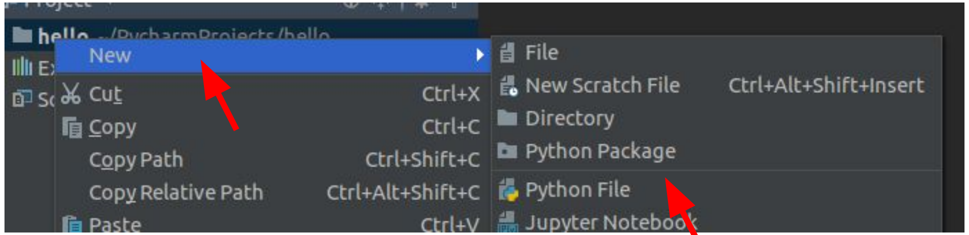
Existing interpreter

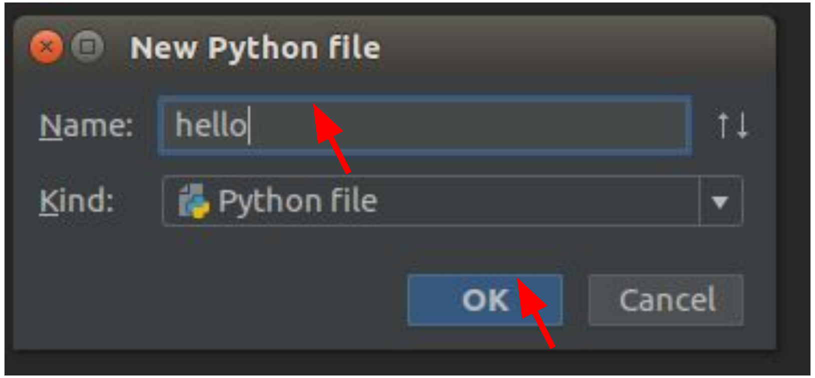
Interpreter: thon 3.6 C:\Program Files (x86)\Python36-32\python.exe

Adding a python file

Right click on the “hello” folder and select “new” → “python file”.
type the name of the file (“hello”) and press OK. We are now ready to type some code!


hello [C:\Users\shv\PycharmProjects\hello] - PyCharm
File Edit View Navigate Code Refactor Run Tools
hello
Project
hello C:\Users\shv\PycharmProjects\hello
External Libraries
Scratches and Consoles



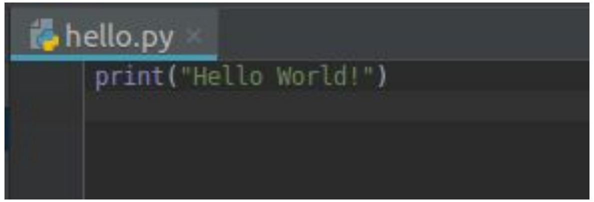


Printing Hello World!

RT Embedded Linux Solutions

Type the following command exactly (python is case sensitive, meaning it differentiates lowercase from uppercase letters!)

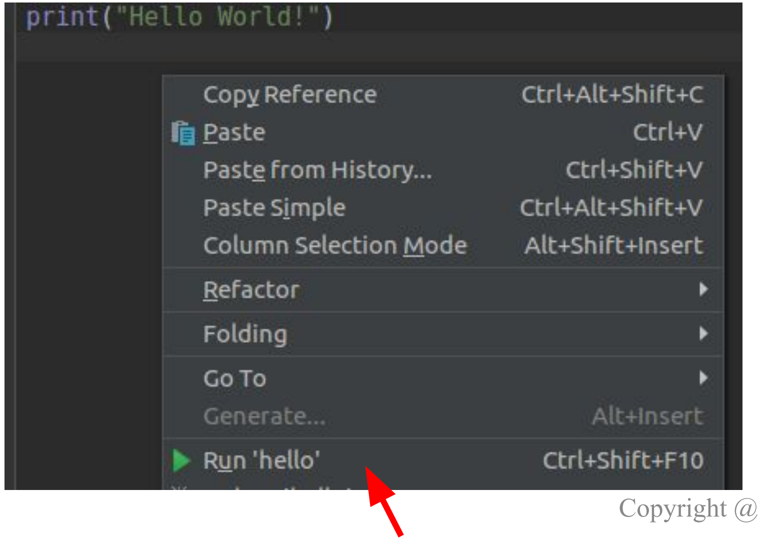
```
print(“Hello World!”)
```



```
print("Hello World!")
```

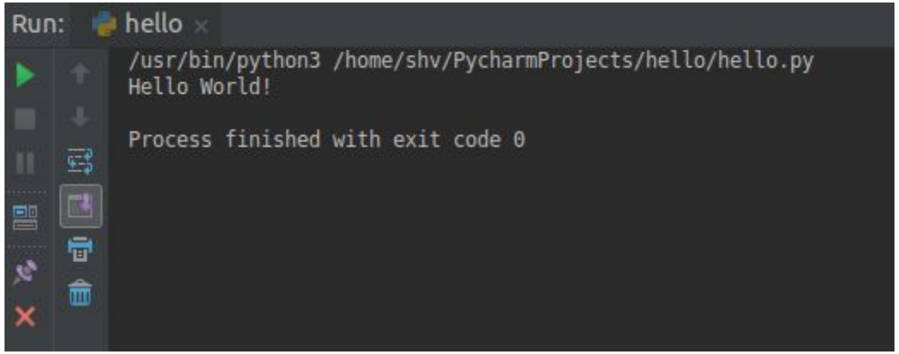
Next, right click on the document and select “run”

If you did everything correct, a new window will be opened showing the result.



```
print("Hello World!")
```

Copy Reference Ctrl+Alt+Shift+C
Paste Ctrl+V
Paste from History... Ctrl+Shift+V
Paste Simple Ctrl+Alt+Shift+V
Column Selection Mode Alt+Shift+Insert
Refactor
Folding
Go To
Generate... Alt+Insert
Run 'hello' Ctrl+Shift+F10



```
/usr/bin/python3 /home/shv/PycharmProjects/hello/hello.py
Hello World!

Process finished with exit code 0
```

Configuring Pycharm

There is no need to save our program, PyCharm autmatically saves any changes we make

Pressing Ctrl+Shift+A will open the “find action” menu, allowing us to type for a setting or an action we want to perform.
For example, typing “line numbers” will allow us to choose the option to show line numbers (select it with the keyboard by pressing down and enter on the item you want to turn on)

Typing mouse zoom will allow us to increase/decrease the text with Ctrl+Mouse-Wheel

You can also press Ctrl+Alt+S to view the settings menu.

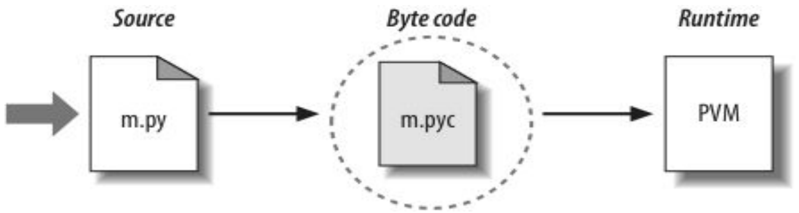
How does Python work?

RT Embedded
Linux Solutions

When you instruct Python to run your script, there are a few steps that Python carries out before your code actually starts.

The code is only partially compiled to a special form of executable, called Python Byte-code.

This executable is not written in the language of your CPU (not a binary file), thus it needs a virtual machine to run. The virtual machine's name is PVM – Python Virtual Machine. This explains the reason Python code may not run as fast as C (or C++) code.



The PVM process is the run-time engine of python. In simple terms, this is the binary the CPU actually reads. Your python code is interpreted through it.

Adding Comments

RT Embedded Linux Solutions

We can add comments to are code. Comments are text which will not be interpreted as a command to run. The purpose of comments are to explain what our program or parts of it is doing.

```
"""
This is a multiline comment, it starts with three quote marks
It can take many lines until we reach three quote marks
"""

''' this is also a multiline comment
using three single-quotes '''

#This is a single line comment. It ends when the line itself ends
print("Hello World!") #comments can be written after the command itself
```

RT Embedded
Linux Solutions

Shebang comment

The shebang comment helps UNIX operating system to execute python files directly by telling the operating system what program to use when trying to execute the .py file directly (Otherwise it might choose the wrong program). (Windows doesn't need the shebang comment because the .py extension is used to identify the file)

The shebang line starts with "#!". Specifically, for python3 you should start your program: **`#! /usr/bin/env python3`**

```
#!/usr/bin/env python3

print("Hello")
```

Printing more than one object

We can send several argument to the print function and it will print them in a single line, separating them by space and ending the printing with a newline (“enter”).

```
print(“The number”, 5, “is greater than”, 7) # Output: the number 5 is greater than 7
```

We can also control what would be printed between each argument instead of space by changing the optional argument sep (short for separator)

```
print(1, 2, 3) #Output: 1 2 3
print(1, 2,3,sep=’X’) # Output: 1X2X3
print(1, 2,3,sep=’’) # Output: 123
```

Additionally, we can change what is printed in the end instead of a newline by changing the optional argument end. (Note: if you change the end argument, sometimes what you print will not appear immediately on the screen, so you may have to set another optional argument flash to True)

```
print(“Hello ”, end=’’)
print(“World”)                    #Output: Hello World
```

Escape Characters

In order to print some characters we can use their escape form:

‘\n’ - represents the newline character (enter)

‘\t’ - represents a tab

‘\r’ - represents carriage return (place the printing marker back at the beginning of the line

‘\b’ - represents backspace (place the printing marker one character backwards)

‘\\’ - prints the ‘\’ character

Working with variables

we can add variables to our program. variables are user-made names (symbols) that can hold any kind of data (or objects, it be more precise)

We can create a new variable just by using its name and using the assignment operator (“=” sign), followed by the value we want to store inside the vairable. we can later change this value using another assignment. Depending on the value, we can perform operations on it. For example, if a variable holds a number we can add it with the addition operator (“+” sign))

```
x = 5
y = 7
z = x + y
print(x, y)
print(z)
```

variable naming: Variables names must be made out of letters A-Z or a-z, digits 0-9 or an underscore. Also, it cannot start with a digit. For this course our convention will mostly use lowercase letters separated with underscores.

Multiple assignments

We can assign multiple values at once using a single = with “lists” of variables and a “list” of values

```
# the following:
a,b,c = 1,2,3

# is the same as:
a = 1
b = 2
c = 3

# a neat trick is to use multiple assignments to swap values of two variables

a,b = b,a    #first python reads the values of b and a on the right side
             #and then assigns them in order to the left side
```

Reading user input

we can read user input using the “input” function. inside the input function we can put a message to the user. when the function is executed the program will wait until the user types something with the keyboard and press “enter”. To save the result we need to assign it to a variable, otherwise the result will be gone.

```
name = input(“What is your name?”)

print(“Hello”, name)
```

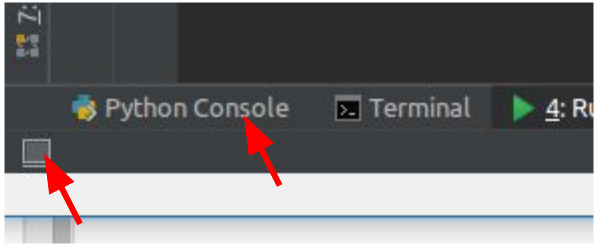
The input result is in text form (string type). We can also convert it to a number (integer type) using the “int” function

```
number = int(input(“Enter a number:”))

print(number+10)
```

Working Interactively inside pycharm

RT Embedded
Linux Solutions

With an open project window in PyCharm, click on the “Python Console” tab on the bottom left (if it doesn’t show click on the square icon to make it appear.



Inside the Python Console Window we are able to execute commands and expressions one by one to see their result immediately (Unlike writing and running a program where all the lines are executed one after another)

Clicking on the “show variables” icon will let us see any variables we create.

Basic Programming Terms

RT Embedded
Linux Solutions

Types

Types represent the kind of data python can operate with. The basic ones are:

int - for integer numbers

float - for floating point numbers (able to store decimal values)

str - string type, represents “text”, a string of characters.

bool - can either be True or False

Other, more advanced types, exist and we will learn about some of them later in this course

Additionally we will learn to create our own types using classes.

Keywords

keywords are reserved words use for commands/data and logic for python. These are:

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

because this words are reserved we cannot create a variable with the same name as a keyword. We will learn to use most of these keywords.

Literals

Literals are text that’s interpreted as a specific value of a specific type:

`123` - digits represents an integer (one hundred and twenty three)

`12.3` - digits with a decimal point represent a floating point number (twelve point three)

`“Hello”` `‘hello’` - typing text inside quotation marks will result in a string (the starting quote must match the ending quote)

(we can also use triple quotes for multi-line strings)

`True` `False` - these are the only available literals for the bool type

None - represents “nothing” , this value is used when there no meaningful value, or if the value is unknown

Objects (Data), Variables & Operators

Objects (Data)

Python stores data in memory. All data in python is stored as an object.
An object can have fields of data and operations it can make.

Variables

Variables are symbols that link to data (objects). It’s important to understand that variables do not contain the object, they only a reference to it. We can have more than one variable referencing the same object. We can also change the variable to “point” to a different object (using assignment operation).

Operators

Operators are symbols used to perform an action on one or more objects in order to create or modify objects.
for example the plus sign (+) when used between two integers will create a new integer based on their addition (2+3 will take the int object 2 and int object 3 and will create a new int object 5)

Functions, Methods & Expressions

Functions

Functions are symbols we can "call" (make python do its "action"). We do so by adding parentheses () after the function name. we can add data inside the function (called arguments) and the result of the function. **print** and **input** are examples of functions.

Methods

Methods are functions that are inside objects. they can be access with a dot symbol and can be called just like functions

Expressions

An expression is anything that results in some data (object). Literals, Variables, Functions, Methods and operator operations are all expressions. An expression can also be made out of other expressions.

Asking for help

RT Embedded
Linux Solutions

The help() function will give us details on how to use an object. For example:

help(123) - will give help on the int type
help(print) - will give help on using the print function
help("+") - will give help on operators

We can trigger the help in Pycharm by pressing Ctrl+Q on the text we want help with.

The dir() function can be used to "explore" objects by printing their fields.

Additional help can be found in the official Python site documentation (Try the tutorial section):
https://docs.python.org/3/index.html

Creating and deleting a variable

Defining a variable is done by typing <var name> = <value>
For example:

```
        var1=10          # this will be an integer
        var2=3.33        # this will be a float
        var3=“200 kg”  # this will be a string
```

To delete a variable use the del command:

```
del var1
```

for example:

```
strHello = “hello world”
print(strHello)
del(strHello)
```

What happens to the data when a variable is deleted

RT Embedded
Linux Solutions

Python uses “Reference Counting” to check which data is usable and which data is not usable.

When an object is assigned to a variable, its RC (reference count) goes up by one.

When that variable is deleted or is re-assigned to a different object, the original object RC is reduced by one.

When an object RC is zero, it’s removed from memory.

In python we have little to no control on object’s lifetime, but we can control the lifetime of variables and other references to objects.

Type conversions

RT Embedded
Linux Solutions

We can “call” types names like functions to “convert” data from one type to another.

```
str(varX)              # expression (varX) will be converted to string

float(varY)            # expression (varY) will be converted to float

int(varZ)              # expression (varZ) will be converted to int
```

We can check the type of an object using the type() function:

```
# output: <type ‘str’>. you can use it, type(var3) == str
type (var3)
```

RT Embedded
Linux Solutions

Operators

Math Operators on numbers

+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Floor division
%	Modulus (returns the division “remainder”)
**	Exponent

```
a=9
b=2

print(“op + : “, a+b)    # out: op + : 11
print(“op - : “, a-b)    # out: op - : 7
print(“op * : “, a*b)    # out: op * : 18
print(“op / : “, a/b)    # out: op / : 4.5
print(“op // : “, a//b)  # out: op // : 4.0
print(“op % : “, a%b)    # out: op % : 1
print(“op ** : “, a**b)  # out: op ** : 81
```

Compound Assignment

we can combine assignment with an operation to get compound assignment. For example typing:

```
a+=5
```

has a similar end result as typing:

```
a = a + 5
```

(A small difference is that python will only read a once in a compound assignment, which may improve its execution speed)

```
#more compound examples:

    a = 5     #  5

    a += 3    #  8

    a -= 4    #  4

    a *= 10   # 40

    a //= 10  #  4

    a **= 2   # 16
```

Math operations on strings

RT Embedded Linux Solutions

We can add (concatentate) two strings to create a new string made of both:

```
“abc” + “123” # returns “abc123”
```

We can multiply a string with a number to add the same string with itself several times:

```
“abc” * 3 # returns “abcabcabc”
```

Note: python is sensitive to the types used. we can’t add a string to ant int. Likewise multuplying a string with a string would make no sense.

```
“abc” + 123  #error
“abc” * “efg” #error
```

String methods

When a variable is a string you can use it’s methods. Methods for example:

`strvar.upper()` - returns a copy with uppercase letters

`strvar.isdigit()` - returns True if all characters are digits

`strvar.islower()` - returns true if all characters are lowercase

This methods only create a modified copy. The can’t directly change the object held by a variable.

To actually change a variable we need to reassign it to the copy:

hw = hw.upper()

Anything typed with quotes has those methods ..

```
var name

hw = "hello world"

#outputs: HELLO WORLD
print(hw.upper())

#outputs: False
print(hw.isdigit())

#outputs: True
print(hw.islower())

#hw = “hello”
hw = “HELLO”.lower()
```

String Formatting with %

We can make a string template with placeholder to be replaced with actual data. For example

```
“My name is %s and I’m %s years old”
```

is a string with two placeholders marked with %s. we can send data to be added in these place holders using the % sign

The code below would print: “`My name is Moshe and I’m 20 years old`”:

```
print(“My name is”,”Moshe”, “and I’m”, 20 “years old”)
```

As will the code below:

```
print( “My name is %s and I’m %s years old”  %  (“Moshe”, 20) )
```

we can use other placeholders to modify the format:

```
“%5d”    % 7   #    7
“%05d”   % 7   #00007
“%.3f”   % 1.5 #1.500
“%7.3f”  % 1.5 #  1.500
“%07.3f” % 1.5 #001.500
```

RT Embedded
Linux Solutions

String Formatting with the format method

Python 3 added an additional way to format a string using the format method. The placeholders for the format are curly braces ({}):

```
“My name is {} and I’m {} years old”.format(“Moshe”, 20)

Inside the braces we can set modifiers for how the data prints:

“{:5d}”.format(20)        #   20
“{:05d}”.format(20)       #00020
“{:.3f}”.format(1.5)      #1.500
“{:7.3f}”.format(1.5)     #  1.500
“{:07.3f}”.format(1.5)    #001.500

We can use < to justify left, > to justify right and ^ to center text:

“|{:<10}|{:>10}|”.format(“Moshe”, “Levi”) #outpus: |Moshe     |      Levi|
```

Indexing, Slicing & Immutability

RT Embedded
Linux Solutions

positive indexes, slices & immutability

```
>>> str1 = “string 1”

>>> str1[2] = ‘z’  # ERROR! slices are immutable. Raises exception in run-time

>>> print str1[2]
‘r’

>>> print str1[:3]
‘str’

>>> print str1[2:]
‘ring 1’

>>> print str1[1:3]
‘tr’
```

0	1	2	3	4	5	6	7
s	t	r	i	n	g		1

negative indexes and slices

RT Embedded
Linux Solutions

```
>>> abc = “abcdefgh”

>>> print abc[-2]
‘g’

>>> print abc[:-4]
‘abcd’

>>> print abc[-3:]
‘fgh’

>>> print abc[-4:-2]
‘ef’

>>> print abc[1:-1]
‘bcdefg’
```

-8	-7	-6	-5	-4	-3	-2	-1
a	b	c	d	e	f	g	h

RT Embedded
Linux Solutions

slice skipping

```
>>> digits= “012345678”

>>> print digits[1:6:2] # slice from 1 to 6, skip size 2
‘135’

>>> print abc[:7:3]
‘036’

>>> print abc[4::2]
‘468’

>>> print abc[-3:1:-1]
‘65432’

>>> print abc[-2:0:-2]
‘7531’
```

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
0	1	2	3	4	5	6	7	8

Advanced Data Types

Collections introduction

Collections can hold more than one object:

List - an ordered modifiable (mutable) collection of object
var = [] # creates an empty list
var = [‘one’, 2, ‘three’, ‘banana’] # create a list with 4 objects

Tuple - an ordered modifiable (mutable) collection of object
var = (‘one’, 2, ‘three’, ‘banana’)

Dictionary - A collection of objects stored by key values (similar to the hash table data type)
var = {} # create an empty dictionary
var = {‘lat’: 40.20547, ‘lon’: -74.76322}
var[‘lat’] = 40.2054 # modify an exsiting value inside a key

Each has its own set of methods. We will learn about each type in the next slides

Lists

- Think of a list as a stack of cards, on which your information is written.
- The information stays in the order you placed it in until you modify that order
- Methods return a string or subset of the list or modify the list to add or remove components.
- Written as var[*index*], index refers to order within set (think card number, starting at 0)
- You can step through lists as part of a loop.

List Methods

RT Embedded
Linux Solutions

- Declare a list
  - list1 = *[]* # Declare an empty list of objects
  - list2 = [3, “a”, 2, True, 5.5] # you can add any number of objects here
  - print list2[4] # output: 5.5

- Adding elements to the list
  - list1.append(*object*), e.g. list1.append(“myItem”)
  - list2.insert(*index, object*) e.g. list2.insert(2, “thirdItem”)

- Changing the list
  - list2[*n*] = *object* e.g. list2[2]=2000
    e.g. list1[0]=“elm num 0”

List Methods (2/2)

RT Embedded
Linux Solutions

- Removing from the List
  - ➢ list1 = [] *# empties contents of card, but preserves order*
- *del list2[n]* *# removes a single card whose index is n*
  - ➢ *list2.remove(n)* *# removes card whose value is n*
  - ➢ *list1.pop(n)* *# removes n and returns its value*

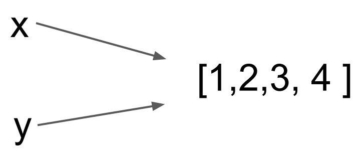
- More list methods
  - ➢ *"examp" in list1* *# checks weather the item "examp" is in the list*
    # returns True or False (boolean type)
  - ➢ *list2.reverse()* *# reverses the list*
  - ➢ *list1.sort()* *# sorts the list*

Lists are mutable object

Up until now, we've only dealt with objects you can't modify. All we did was create a copy. for example: 1+2 does not change 1 into becoming 3, instead it creates a new int object with the value 3. This is why we had to reassign variables if we wanted to remember the new result.

But now we have lists, which are mutable objects, meaning we can modify its inner data. For example:

```
x = [1,2,3]
y = x

x.append(4)
print(x)
print(y)
```



Both print functions would print [1,2,3,4] this is because x and y point to the same object in memory and the append method modified the inner data of that object.

Object identity

We can tell the difference between each object using its memory location. using the id() funciton we can print the object memory location:

```
x = [1,2,3,4] #create a list
y = x           #make y point to the same list as x
z = [1,2,3,4] # create a new list

print(id(x)) #will print some number
print(id(y)) #will print the same number as above
print(id(z)) #will print a different numbers
```

It's important to note that even though the list in x and z has the same contents, they point to different objects.

```
x == z              # will print True because they have the same content
x is z              # will print false because they are not the same object
x.append(5)
print(x)            #will print [1,2,3,4,5]
print(z)            #will print [1,2,3,4]
```

Tuples

Like a list, tuples are an ordered collection of objects. while we usually use parenthees () to idnetify a tuple, we can simply use just commas (,)

```
x = 1,2,3
y=  (1,2,3)
```

If you want a tuple of a single item, you can use the following: z = (,1) # tuple containing a single element 1

Tuples are immutable – once created, unchangeable (we can however apply operations that create a copy)

Advantages of being immutable: safe from accidental changes, might be faster to process
Advantages of being mutable: makes it easier to write code that modified data

Slices apply to ordered collections

Strings, are also an ordered immutable collection just like tuple, so many of the operations that are available on strings can be used on tuples and vice versa. Likewise, many of the operations that work on ordered collections, can work on both strings lists and tuples.

Here's a few:

concatenation	(1,2,3) + (4,5,6) # (1,2,3,4,5,6)
multiplication	(1,2,3) * 3 # (1,2,3,1,2,3,12,3)
reversed	(1,2,3) # returns a reversed copy (3,2,1)
length	len((55,66,77)) # returns the number of elements: 3
max	max((55,77,66)) #returns 77
min	min((55, 77, 66)) #returns 55
indexing	(10,20,30,40,50)[0] # returns 10
slicing	(10,20,30,40,50)[3:1:-1] #returns (40,30,20)

in-place vs copy-operations

When working with a mutable objec it's important to know if an operations will modify the inner data (mutate it) or if it will instead produce a modified copy. For example, the list.reverse() method will modify the actual list and reverse it. We call this kind of operations "in-place".

```
x = [1,2,3]
y = x.reverse() #revese changes in place and returns None, so y is None
print(x) # will print [3,2,1]
print(y) # will not print anything
```

the reversed function, on the other hand, does not modify in-place and instead produces a reversed copy. (To know which operation is in-place and which is not you need to read its documentation, i.e. read its help)

```
x  = [1,2,3]
y = reversed(x) # reversed returns a reversed copy, and doesn't modify x
print(x) # will print [1,2,3]
print(y) # will print [3,2,1]
```

Dictionaries

Dictionaries are sets of key & value pairs. This allows you to identify values by a descriptive name instead of order in a list
Keys are unordered unless explicitly sorted. a dictionary item is written in the form of key:value

```
x = {‘firstname’: ‘moshe’, ‘lastname’:’levi’, ‘id’:123456}
```

keys must be an immutable object, but can be any immutable objects (they can be tuples,string,numbers, but not lists

Keys are unique:

```
var[‘item’] = “apple”
var[‘item’] = “banana” #override the values in key ‘item’
print var[‘item’] # prints just banana
```

```
# you can make the dictionary from any value to any value
  var[‘pie’] = 3.14
  var[3.14] = True

  del var[‘pie’] # you can delete a specific item with ease
```

more dictionary operations

You can check if a key exists using the “in” operator:
x = {1:”aaa”, 2:”bbb”, 3:”ccc”}

print(1 in x) # will print True
print(4 in x) # will print False

if you try to access a key that doesn’t exist you will get an error: print(x[4]) #error

we can use the get method to try to access a value ina key, and if the key doesn’t exists a default value will be used:
print(x.get(‘name’, ‘unkown’)) #will print “unknown”
print(x.get(1, “unknown”) #will print “aaa”

you can get a list of all the keys by typing: list(x.keys())
likewise you can get a list of all the values by typing: list(x.values())

Header-line and Block Model

Header-line and Block Model

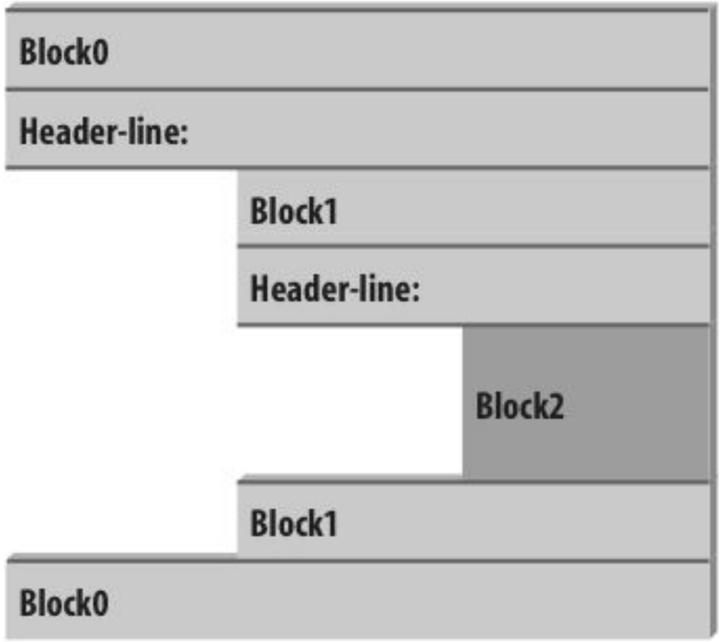
RT Embedded
Linux Solutions

In any programming / scripting language we are given the choice to brunch to different parts of code.

Python defines a well-structured block boundaries and detects them automatically. Every branching statement has a header-line, and a block of code.

This intro is the same for:

- conditions
- loops
- functions
- classes
- exception statements



RT Embedded Linux Solutions

Header-line and Block Model (2)

Unlike most mainstream Programming:

- There are no braces or "begin/end" delimiters around blocks of code in Python such as *{* or *}* in C/C++/C#/Java. All done with <TAB> character.

```
#include <stdio.h>

void main(int argc, char **argv)
{
      printf("hello world");

     if (argc == 1)
           printf("no params");
      else
      {
           printf("%d params", argc – 1);
           printf("exit successfully");
      }
}
```

```
#!usr/bin/python
import sys
print "hello world"
if len(sys.argv) == 1:
    print "no params"
else:
    print "%d params", len(sys.argv)-1
    print "exit successfully"
```

Header-line and Block Model (3)

RT Embedded Linux Solutions

Unlike most mainstream Programming, Python statements are not normally terminated with semicolons *;* rather, the end of a line marks the end of the statement coded on that line.

```
#include <stdio.h>

void main(int argc, char **argv)
{
      char name[100];

      printf("please enter your name");
     scanf("%s", &name);
     printf("welcome %s", name);
}
```

```
#!/usr/bin/python


name = raw_input("please enter your name")
print "welcome %s", % name
```

Header-line and Block Model (4)

RT Embedded Linux Solutions

Unlike most mainstream Programming:

- All compound statements in Python follow the same pattern:
  - a header line terminated with a colon, followed by one or more nested

```
# incorrect example

    x = 'SPAM' # Error: first line indented
if 'rubbery' in 'shrubbery':
    print(x * 8)
        x += 'NI' # Error: unexpected indentation
        if x.endswith('NI'):
                x *= 2
           print(x) # Error: inconsistent indentation
```

```
# correct example

x = ‘spam'
if 'rubbery' in 'shrubbery':
    print(x * 8)
    x += 'NI'
    if x.endswith('NI'):
        x *= 2
        print(x) # Prints “spamNIspamNI"
```

Conditions

Conditions

RT Embedded
Linux Solutions

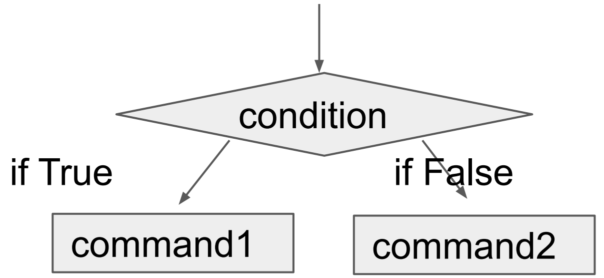
Normally the code flows from one command to the next. But sometimes in our code we need to have decision-making that will choose one path or another

In simple terms, Python conditional statements selects actions to perform / ignore.

Those statements may contain one or more nested statements, and even more conditional branching.

Python has the following statements:

if
elif
else



Conditions (2)

The Python if statement is typical of if statements in most procedural languages.
It takes the form of an if test, followed by one or more optional elif ("else if") tests and a final optional else block

```
if <test1>:                 # if test
        <statements1>     # Associated block
elif <test2>:               # Optional elifs
        <statements2>
else:                       # Optional else
        <statements3>
```

Truth Test

RT Embedded Linux Solutions

The if and elif statements checks whether a condition evaluates to true.

But what python considers true ?

- Any nonzero number or non-empty object is true.
- Zero numbers, empty objects, and the special object None are considered false.
- Comparisons and equality tests are applied recursively to data structures.
- Comparisons and equality tests return True or False (custom versions of 1 and 0).
- Boolean *and* and *or* operators return a *true* or *false* operand object.

Relational operators

Relational operators are boolean operators (they return either True or False) the can check the difference between two values:

```
x=5
y=7
text1 = “hello”
text 2 = “jello”

print(x < y) # True
print(x == y) # False
print(x <= y) # True. x is greater or equal to y

print(text1 == text2) #False
print(text1 != text2) #True, they have different content
```

List of operators:

< less than

> greater than

== equals

>=greater or equals

<= less or equal

!= different

X is Y - X is same object as Y

X in Y - X value is inside Y

(or X is key if Y is a dictionary)

Logical operators

RT Embedded
Linux Solutions

Logical operators are boolean operators that can combine other boolean statements using truth tables:

S1	S2	S1 and S2	S1 or S2
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Additionally, the not operator can reverse a True result into a False, and a False result into a True

Condition Examples:

```
x = int(input(“enter a number for x:”))

if x >= 0 and x >= 100:
        print(“x is between 0 and 100”)

#We can also get the same result by typing:
if 0 <= x <= 100:
        print(“x is between 0 and 100”)

#We can also use the “in” operator for collections
if x in 0,1,2,3,4,5:
        print(“x is between 0 and 5”)

x = input(“enter an integer from 0-10:”)
if not x.isdigit() or not 0<= int(x) <=10:    #check if x is not a valid int, or, if it’s a valid int, check if it’s in range 0-10
        print(“not a valid input”)
else:
        print(“good”)
```

short circuiting

python uses optimization when calculating statements made of “and”s and “or”s

A and B and C   # python will not read/execute B unless A is True. likewise will not read/execute C unless A and B is True

A or B or C # python will not read/execute B unless A is False. likewise will not read/execute C unless A and B is False

we can use this to add a condition that requires another condition to be legal:

```
x = {‘a’:”good”, ‘b’:”bad”, ‘c’:”good” }
k = input(“enter key”)

if x[k] == “good”:                      # x[k] will fail if k is not ‘a’,’b’ or ‘c’
        print(“good choice!”)

if k in x and x[k] == “good”:           # x[k] cannot fail because it’s only checked if “k in x” returns True.
        print(“good choice!”)
```

What about switch / case ?

RT Embedded
Linux Solutions

- you might be interested to know that there is no *switch* or *case* statement in Python that selects an action based on a variable's value
- Instead, multi-way branching is coded either as a series of *if/elif* tests
- Another possibility is using an advanced built-in data-type, e.g. indexing dictionaries or searching lists.

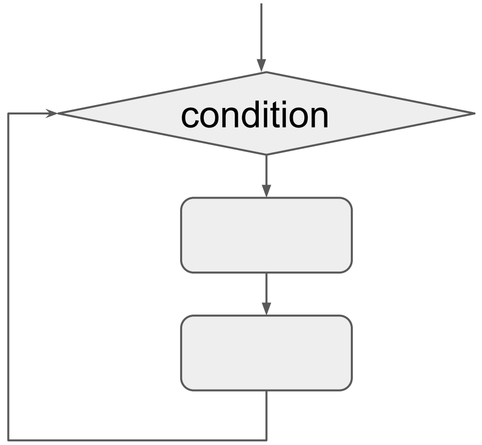
```
if <expression1 >:
            statement1
elif <expression 2>:
            statement2
elif <expression 3>:
            statement3
…
else:
            statement
```

Loops

Loops

RT Embedded
Linux Solutions

Like conditions, loops allow us to control the flow of the program. Unlike conditions, loops allow us to repeat the same lines of code many times. usually when there’s a repeating pattern, or any time we need to do some action indefinite number of times, we can use a loop.



While loops

As a rule of thumb, a loop will usually have 4 elements to it:

initialization:	setting values to be used for the conditions, placed before a while header
condition:	checking whether or not to continue the loop or not, will be placed inside a while “header”
work:	the actual command we want to repeat, placed inside a while block
increment:	any kind of change to the values that are used to test for the condition. placed inside a while block

Example:

```
i = 0                # initalization: set i to 0
while i < 10:        # condition: only start/continue the loop while i is less than 10. quit when i is greater than 10 (11)
    print(i)         # work: prints the current value of i
    i+=1             # increment: adds 1 to i (making to go from 0 to 1 to 2 and so forth
```

If we want a loop to continue without stopping we can say:

```
while True:                #this loop will print “HA HA HA HA HA HA “... endlessly
    print(“HA ”)
```

using while loop to iterate over collections

RT Embedded Linux Solutions

We can use indexing and while loops to iterate over a collection:

```
names = [‘david’, ‘moshe’, ‘aaron’, ‘lea’ ]

i = 0
while  i < len(names):        #we utilize the fact that legal indexes are from 0 to one number below the list length
                              #so for list names with 4 elements the legal indexes are from 0 to 3
        print(name[i])        #slices don’t have to be integer literals, we can use any expression which return an int
                              #in our case we use the variable i which is interpreted to be the int it points to

        i+=1                  #set i to be the next index
```

#this loop will print all the names one after another, each in a new line

For loops

RT Embedded Linux Solutions

For loops make it easier to iterate over a collection without needing to add initialization, condition, or increment. so our code can be made a lot simpler by only containing the “work”.

```
names = [‘david’, ‘moshe’, ‘aaron’, ‘lea’ ]

”””
x is a variable that for will automatically create and will automatically assign each value in order
from the names list. Note: x is not a keyword, you can rename it to whatever you want.
”””

for x in names:
        print(x)                # each iteration x will be assigned a different name

#this loop will print all the names one after another, each in a new line
```

For loops for other collections

We can also iterate over a dictionary, python will understand this as iterating over the dictionary’s keys:

```
data =  {‘first-name’:’moshe’, ‘last-name’:levi, “age”:20}

for k in data:              #every iteration, k will be assigned the next key in order: “first-name”, “last-name”, “age”
          print(k, ‘=’, data[k])

”””
Output:
first-name = moshe
last-name = levi
age = 20
”””
```

We can even iterate over a string, which will iterate over each of its characters:

```
for x in “abcdefg”:
          print(x)                    #will print the characters a, b, c, d, e, f, and g each in a new line
```

Range function

sometimes we just want to make the loop runs specific number of times. we can use the range functions to create an ordered collection for our loops. Range works similar to slices. range only works with integers

range(10) → will produce a collection of 0 .. 9

range (5,10) → will produce a collection of 5 .. 9

range (5, 10, 2) → will produce the collection of values 5,7,9

We can then use range for our loops:

```
for x in range(10):
        print(x)

# wil print 0 .. 9 each in its own line
```

For loop over pairs

If we have a list where each item contains a tuple pair:

```
x = [ (1, 10) , (2, 20), (3, 30), (4 ,40) ]
```

We could iterate it by using the following code:

```
for v in x:
        print(x[0], x[1])
```

But we could also use multiple assigmnet:

```
for a,b in x:
        print(a,b)
```

enumerate

for loops normally work with the inner values directly, this means we can’t access the index and can’t modify the collection.
In order to do so we can use the enumerate function, to create a list of pairs of (index, value) :

```
x = [‘a’, ‘b’, ‘c’]
print(   list(enumerate(x)) ) # will print [(0, ‘a’), (1, ‘b’), (2, ‘c’)]

for i,v in enumerate(x):
        print(i, ‘=’, v)                #will print in order 0=a , 1=b, and 2=c each in  its own line
        x[i] = ‘Z’              #will overwrite the values of the list in x to be ‘Z’

print(x) # will print [‘Z’, ‘Z’, ‘Z’]
```

Control Statements - Continue

RT Embedded Linux Solutions

using control statements we can change how a loop flows in each iteration or to end a running loop:

continue: skip the current loop iteration and move to the next one

```
for x in range(100):
        if x % 2 == 0:          #if x is an even number
                print(x)        #print it
        else:                   #if x is an odd number skip to the next iteration
                continue
        print("----")           # this line will only be printed if continue is not executed
```

Control Statements - break

RT Embedded
Linux Solutions

break: exits the loop and moves to the line after the loop block. useful for exiting a loop for finding an element

```
for x in range(1000000):
        if x%2==0 and x%5 ==0 and x%7==0:
                break

print(“the first number from 1..1000000 to divide by 2,5 and 7 is”, x)
```

when we will learn about functions, we will be able to use the return statement to exit a function, and as a result, exit any running loop inside that function.

Functions

Functions

- Functions are a nearly universal program-structuring device. You may have come across them before in other languages, where they may have been called *subroutines* or *procedures*.

- To create a function, use the keyword *def*, followed by header-line and (): and the function's body. You can also add parameters as you please.

- The keyword *return object* sends the result of the function back to the caller.

Functions

- Functions are a nearly universal program-structuring device. You may have come across them before in other languages, where they may have been called *subroutines* or *procedures*.

- To create a function, use the keyword *def*, followed by header-line and (): and the function’s body. You can also add parameters as you please.

- The keyword *return object* sends the result of the function back to the caller.

Function Examples

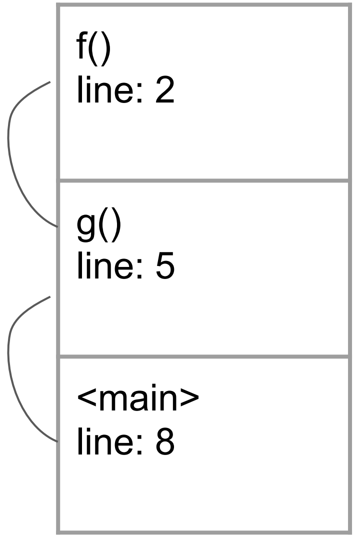
```
def plus_five(x):
        return x + 5

print( plus_five(10) ) # will print 15


def add_ten(x):
        for i,v in enumerate(x):
                x[i] = v + 10


add_ten( [1,2,3,4,5]
print( x ) # will print [11,12, 13, 14, 15]
```

RT Embedded
Linux Solutions

The Stack

Python uses a stack system to keep track of which functions is called by which, it also keeps track of what variables belong to which function. when a function is called, python adds a new “Frame” on the stack with information for that function. when a function finished, python removes its frame and moves to the frame below it to know what to return to.

```
def f():
        print('f')

def g():
        f()             # calling f from a g function frame
        print('g')

g()                     #calling g from a <main> frame
```



Modules

Importing existing modules

RT Embedded
Linux Solutions

If we want additional specific functionality, we can load code from other modules containing the functionality we need.
For example if we want advanced math operations (like square root, or a representation of PI) we can import the math module:

```
import math # adds a module object named math under a variable named math into our code

# we can access the elements of a module using the dot operation:

x = math.sqrt(9)       #access the sqrt function and perform the square root of 9
print(x)               #prints 3.0

print(math.pi)         #prints 3.141592653589793
```

Different ways to import

We can rename the module we import using the “as” keyword:

```
import math as m
print(  m.sqrt(9)  )
```

We can also import just the fields we need without the module name itself:

```
from math import sqrt, pow, pi

print ( sqrt(pi))
```

A shortcut for importing all the fields of a module is to use an asterisk:

```
from math import *

print( sqrt(pi))
```

The random module

RT Embedded
Linux Solutions

The random module lets us perform randomization in our code:

```
import random

# x will be a random number from 1,6 (including 6)

x= random.randint(1,6)

x = [1,2,3,4,5,6,7,8,9,10]

random.shuffle(x)

# shuffles x in place, x will contain the elements 1 to 10 in a random order
```

Creating our own modules

Every python file we create can be treated as a module of its own.

<project directory>/dice.py

```
from random import randint
def roll_d6():
        print(“you rolled a”, randint(1,6))

def roll_d20():
        x = randint(1,20)
        if x == 1:
                print(“critical failure!”)
        elif x==20:
                print(“critical hit!”)
        else:
                print(“you rolled a”, x)
```

<project directory>/dnd.py

```
import dice

dice.roll _d6()
dice.roll _d6()

dice.roll_d20()
```

import process, __main__ separation

RT Embedded Linux Solutions

When we call import for a module, python looks for a .py file with the same name. it then runs the entire file from start to finish, any variable (including functions) which is created is stored as a field inside that module.

If we want to have some of the module code to not run when imported we can place it inside a special condition. __name__ is a special built in variable. it's set to "__main__" only if the code is not imported

```
#file name: a.py

N = 10
def f():
        print("N =", N)

print("START")

if __name__ == "__main__":
        x = N + 5    #x does not exist inside the module
        print("this will only work if you run this code")
        print("not if you import it")
```

```
import a        #will print "START"

print(a.N)      #prints 10
a.f()           #prints N = 10

print(a.x) # error: ... a has no attribute x …
```

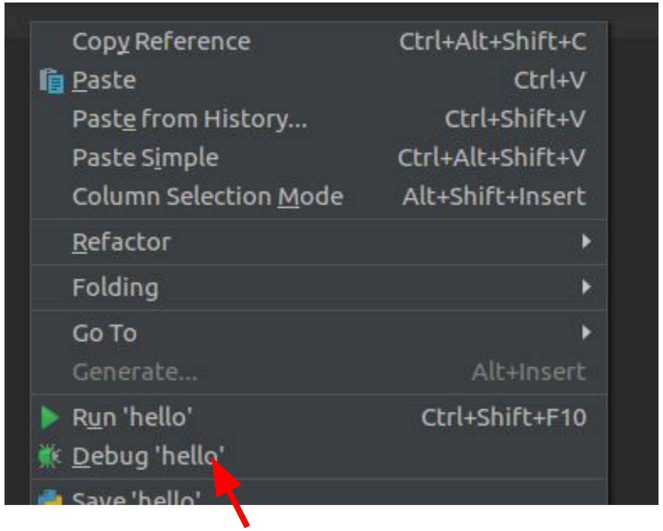
Debugging in Pycharm

Debugging

RT Embedded Linux Solutions

If we have bugs in our code (something doesn't work correctly), we can run our program in debugging mode in order to follow its execution while reading its state (the values in it variables and the state of the call-stack)

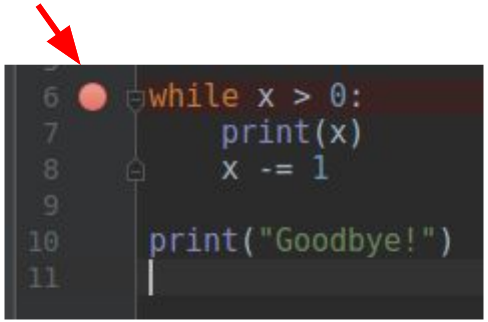
We can start debugging by right clicking the document and selecting debug (bug icon)


Copy Reference Ctrl+Alt+Shift+C
Paste Ctrl+V
Paste from History... Ctrl+Shift+V
Paste Simple Ctrl+Alt+Shift+V
Column Selection Mode Alt+Shift+Insert
Refactor
Folding
Go To
Generate... Alt+Insert
Run 'hello' Ctrl+Shift+F10
Debug 'hello'
Save 'hello'

Breakpoints

If we debug without setting breakpoints, our program will run like normal and we won't get a chance to examine it. Adding breakpoints allows us to pause the execution in specific lines.

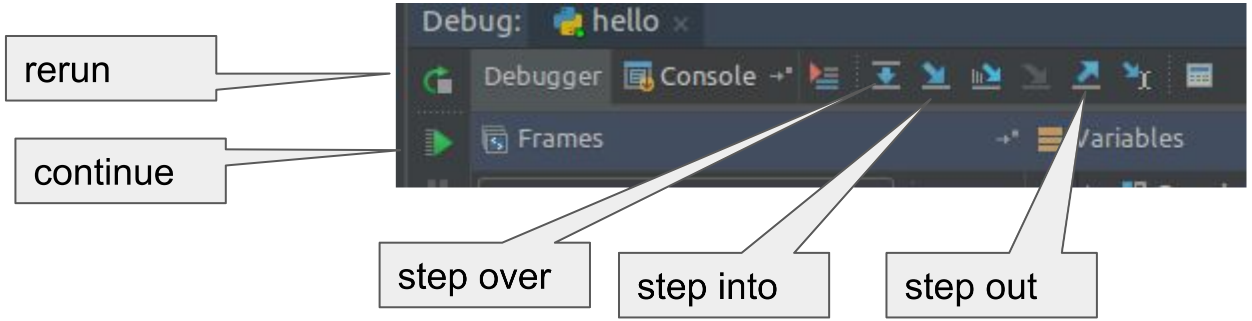
To add a breakpoint click on the area to the left of the line you want to break at (click between the line number and the code). A red circle will appear to denote the breakpoint exists. we can click it to remove the breakpoint.



```
while x > 0:
    print(x)
    x -= 1

print("Goodbye!")
```

Stepping (1/2)

During debug, a special debug pane will open and will contain the following buttons



Stepping (2/2)

RT Embedded
Linux Solutions

rerun - restart the program

continue - keep executing lines until you reach the next breakpoint

step over - execute a single line, if it’s a function, execute it completely

step into - execute a single line, if it’s a function, go inside it and pause over the first line of the function

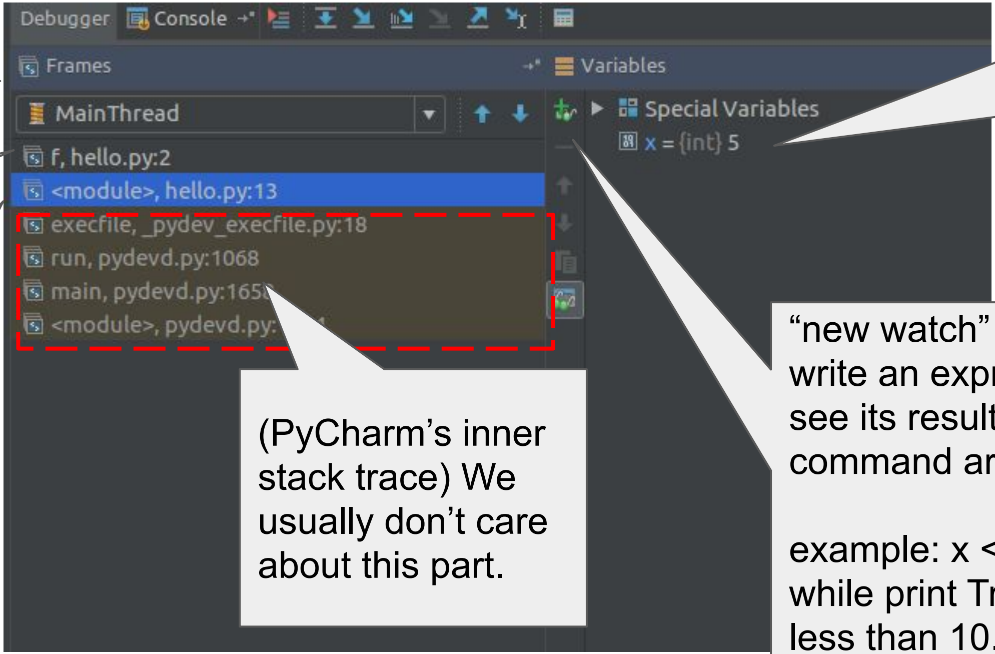
step out - keep executing lines until you exit the current function you are in

The Debugger Pane

the Stack Frame

function f waiting on line 2.

main module waiting in line 13 for function f to finish



```
f, hello.py:2
<module>, hello.py:13
execfile, _pydev_execfile.py:18
run, pydevd.py:1068
main, pydevd.py:165
<module>, pydevd.py:
x = {int} 5
```

(PyCharm's inner stack trace) We usually don't care about this part.

variables inside selected frame (main module)

“new watch” allows us to write an expression and see its result changing as command are executed.

example: x < 10
while print True while x is less than 10.

Files

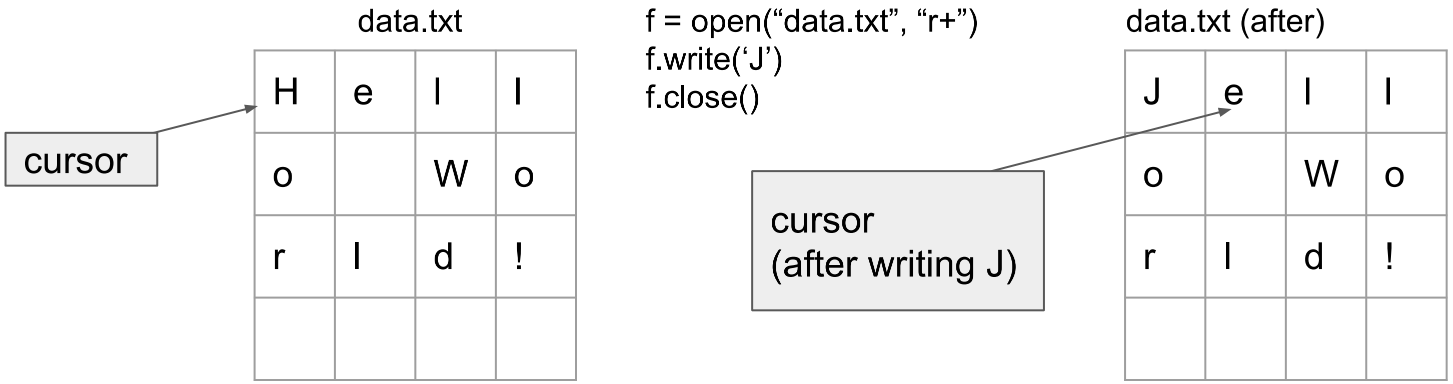
Files

We could consider a file to be a collection of bytes (bytes are small units of data that can each hold a number from 0 to 255)
Any number of bytes can be interpreted as some sort of value depending on how we read it.

In order to read/write to a file we need to ask permission from the operating system to open that file with the correct permissions.

After we open a file we get access to the file data one at a time using a “cursor” pointing to the current data.

Whenever we read or write, the cursor is automatically moved forward.



Files

- Files are manipulated by creating a file object

  f = open("points.txt", "r")

- *The file object then has new methods*

  print f.readline() # prints line from file

- Files can be accessed to read or write

  f = open("output.txt", "w")
  f.write("Important Output!")

- Files are iterable objects, like lists

Files example

```
>>> f = open('/tmp/workfile', 'r+') #read and write
>>> print f  #displaying the content

>>> f.read()
'This is the entire file.\n'


>>> f.write('This is a test\n')


>>> f.close()
```

Files:
No need to import anything....
A built-in type in python.

- **file** / open(name[, mode [,buffering]])**:** opens a file, returns the file object.

name: the file’s name (and full path if needed).
mode: how to open the file ..
‘r’ = read, ‘w’ = write, ‘a’ = append,
‘r+’ = read & write, must exist, ‘w+’ = read & write + overwrite,
‘a+’ = read & write, append only.
buffering: specify the amount of size to buffer.

- close()**:** closes the file. (writes any changes, and can no longer operate on it)

- read([size]): reads the file’s content.

size: if not specified, reads all the file. If specified, reads the amount of bytes.

Files:(2)

RT Embedded
Linux Solutions

- write(str): writes the string to the file.
- flush(): makes sure that the file is updated.
- tell(): tells what byte am I in the file.
- seek(offset [, whence]): reposition in the file

offset: how many bytes to go from the offset (default – beginning of the file)

whence: determines the relative offset ..

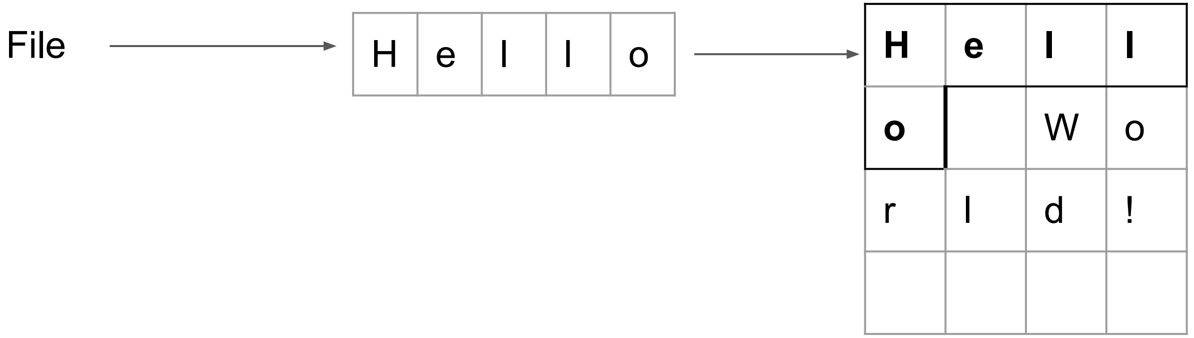
- 0 – beginning of the file.
- 1 – from current position.
- 2 – from the end of the file..

Understanding Buffering

RT Embedded Linux Solutions

In order to work faster, when working with a files, we don’t actually access its data directly. instead a special memory called buffer is used. the buffer is filled with some of the data inside the file and any read/write operation is only done to the buffer. Only after we use the close() method or the flush() method will the changes be written to the actual file.

(Additionally, some actions may cause the buffer to flush the data. For example printing ‘\n’ may cause a flush operation. Also, reading/writing more than the buffer can hold will cause it to be filled with the next available data)



working with bytes

In order to work with binary files we first must understand the basics of bytes.
Bytes are the building-block of every type of data we can think of. A single byte is a number from 0-255.
we can convert a tuple of numbers from 0-255 into a byte object by typing:

```
bytes( (1,2,3,4) ) # will create a bytes object with bytes of values 1, 2,3 and 4
```

bytes objects are immutable and ordered just like a tuple. we can access a single byte number using indexing. we can also use slicing and most other operations we could do on a tuple. we can also convert bytes into a list or a tuple using the list() or tuple() function.

we can also type bytes directly using strings starting with b. each character represents its ascii value as number. we can also use the escape form ‘\xNN’ to type hexadecimal values.

```
x = b’ABCD\x01\x02’
print( tuple(x) ) # prints ( 65, 66, 67, 68, 1, 2) 65 is the ASCII table value of ‘A’, 66 is for ‘B’ and so forth
```

We can check what is the ASCII value of a character by using the ord() function:

```
print(ord(‘A’)) # prints 65
```

String formats

RT Embedded
Linux Solutions

In order to store text, the computer needs some way to converts numbers stored in bytes into characters. There are several ways to do this and we call each way the string format. A format is a table explaining what character to use for each number. By default, most of python's functions and operations use the UTF-8 table:

https://www.fileformat.info/info/charset/UTF-8/list.htm

A common simpler format, is the ASCII format (UTF-8 is a superset of the ASCII table, meaning that every characer in ASCII has the same value as that character in UTF-8):

https://www.asciitable.com/

Encoding and Decoding

When opening a file in text mode, it’s by default opened in text mode and in utf-8 encoding.
If we know the file uses a different encoding we can list it when opening the file:

f = open(“data.txt”, encoding=”ascii”) # open the file in ascii encoding instead of utf-8 encoding

We can use bytes’ decode method to convert it to a string:
x = b’\x65\x66\x67\x68’

s = x.decode() #decode x using UTF-8 encoding. s will equal “ABCD”
s2 = x.decode(‘ascii’) #decode x using ASCII encoding. s2 will equal “ABCD”

We can also use str’s encode method to convert a string into bytes
“ABCD”.encode() # converts to bytes using UTF-8 encoding
“ABCD”.encode(‘ascii’) #convert to bytes using ASCII encoding

A list of available encodings can be found at:
https://docs.python.org/3/library/codecs.html#standard-encodings

converting ints into bytes

When storing an integer in bytes we need to consider two things:
byte-size: how many bytes the number should take
byte-order: Also known as endianness. in what order to store the bytes containing the number. There are two common orders: little-endian and big-endian.

For example the number 258 is built of the bytes 1 and 2.
If we store it in size 2 and order ‘big’ it would be : (1,2)
If we store it in size 2 and order small it would be: (2,1)
If we store it in size 4 and order ‘big’ it would be : (0,0,1,2)
If we store it in size 4 and order small it would be: (2,1,0,0)

int.to_bytes and int.from_bytes

RT Embedded
Linux Solutions

int.to_bytes(size,order) method can convert an integer into its byte form. order needs to be ‘little’ or ‘big’

i = 1000

b = int.to_bytes(8, ‘little’) # will convert the number into 8 bytes stored in little-endian order\

likewise we can use int.from_bytes(bytes, order) to converts bytes into an integer

data=b’\x01\x02’

num = int.from_bytes(data, ‘little’) # will store it as 1,2 in little which is the number 513
num = int.from_bytes(data, ‘big’) # will store it as 1,2 in little which is the number 258

opening a file in binary mode

RT Embedded
Linux Solutions

To open a file in binary mode add the ‘b’ character to its mode:

f = open(“data.bin”, “r+b”) # open the file in binary mode and in “read plus” mode

We can then write and read bytes data from it as though we were reading characters
i = 100
x = f.read(4) # x now holds 4 bytes

f.write(b’\x01\x02\x03) #write the bytes 1,2 and 3 in order
f.write(i.to_bytes(8, ‘big’) # write the number in i in 8 bytes in big endian order

f.write(“hello world”.encode()) # convert the text “hello” into its bytes equivalent in UTF-8 encoding and write those bytes

Designing a file protocol

When working in files (especially binary files) it’s important to note what our protocol for that file is.

A protocol is a set of rules and definitions. For our case, a protocol means how the file is built and how should we approach when reading or writing into it.

for example, we could say our protocol is that our file data.bin contains two integers each of size 8 and each in big order. So in order to read the data we could type:

```
f = open(“data.bin”, “rb”)
num1 = int.from_bytes(f.read(8), ‘big’)
num2 = int.from_bytes(f.read(8), ‘big’)
```

And to write into it we could type:

```
f = open(“data.bin”, “wb”)
f.write( num1.to_bytes(9, ‘big’)
f.write( num2.to_bytes(9, ‘big’)
```

Alternatively, we could use a simpler protocol where the data is stored in text and each line in that text is a number. so we could read the string of each line and convert that line into an int.

The JSON protocol

Instead of making our own protocol, we could an existing one such as the JSON protocol.
Then JSON protocol converts data into text and text into data. This protocol is now aware of the python language and so it cannot store complex python objects, and most data will be converted into lists, dictionaries, strings, ints and floats.

to save json to a file we need to open one in text mode.

```
import json
x = { ‘a’:100, ‘b’: 200 }

f = open(“data.json”, “w”)
json.dump(x, f)              #write data x into opened file f
f.close()

f = open(“data.json”, “r”)
x = json.load(f)             # load all the string data from f as interpreted by json into python data
print(x)                     #should print a dictionary data of { ‘a’:100, ‘b’:200}
f.close()
```

The pickle protocol

RT Embedded
Linux Solutions

Pickle is a python protocol for converting any python object into bytes. it uses the same set of methods as JSON, except that it need an open binary file (not text file) in order to work.

```
import pickle

x = { ‘a’: [1,2,3], ‘b’ : [4,5,6] }
f = open(“data.pickle”, “wb”)
pickle.dump(x, f)                    #write x in pickle binary into open file f
f.close()

f = open(“data.pickle”, “rb”)
x = pickle.load(f)                          #read pickle binary data from file f and convert it to a python object
f.close()
print(x)                                    #should print the dictionary data of { ‘a’: [1,2,3], ‘b’ : [4,5,6] }
```

“with” header

Instead of having to remember to close a file every time we open it, we can open a file inside a “with” header and python will automatically close it when we exit the block:

```
with open(“data.txt”, “r+”) as f:
        # . . .
        # do some operations on f
        # . . .

# f will be automatically closed before we reach this line.
```

(“with” header will also be useful with other objects which we can open and close such as sockets and database connections, which we will discuss about in the future)

Errors and Exceptions

Different kinds of problems

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some.

We will distinguish between the following kinds of problems in our running code:

syntax errors	python could not understand the code the the programmer wrote (made a syntax mistake). These are relatively easy to fix, rewrite the lines reported in the error
exceptions	Something unusual happened that python could not handle by itself. To fix these you need to find the appropriate area to handle the exception and write code that handles it
bugs	the programs runs without crashing, but it's not doing what we want it to do. These are the hardest to fix because the require debugging the code to understand what went wrong and where

Syntax Errors

RT Embedded
Linux Solutions

- Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print 'Hello world'  # Missing ':'
  File "<Example>", line 1, in ?
    while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

Exceptions

Errors detected during execution are called exceptions and are not unconditionally fatal

You will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, and result in an error.

An example of an error would be to try and open a file that doesn’t exists for reading:

```
f = open(“fake.txt”)

#FileNotFoundError: [Errno 2] No such file or directory: 'fake.txt'
```

Capturing an exception

- Check for type assignment errors, items not in a list, etc.
- Try & Except

```
    try:
                    a block of code that might have an error
    except:
            code to execute if an error occurs in "try"
```

- Allows for graceful failure

Capturing an exception (2)

RT Embedded
Linux Solutions

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops!  That was no valid number.  Try again..."
...
```

Capturing an exception (3)

RT Embedded Linux Solutions

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

Capturing an exception (4)

mbedded
Solutions

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print("I/O error({0}): {1}".format(e.errno, e.strerror))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

We can use the “as” keyword to save the exception object

exception objects have fields we can access to get more information on the error

We can handle more than one type of error, individually by adding more “except” headers.

We can handle all other errors by using an except without specifying the type used.

raise keyword

RT Embedded
Linux Solutions

we can generate our own errors from one of the existing error types:
https://docs.python.org/3/library/exceptions.html

to do that we can call an exception type like a function to create a new exception object. we can then raise that object using the raise keyword:

```
e = ValueError("Not the right value") # the string inside ValueError holds the message to show to the user
raise e                               #  start the whole exception process. if not caught with except the program will crash
```

We can also use raise by itself inside an except block to re-raise the same exception we caught (see previous page last line). This is called a catch-and-throw. it's used when we only want to handle the problem partially while still wanting for the exception to be caught by an outer scope.

Object Oriented Programming

Object oriented programming

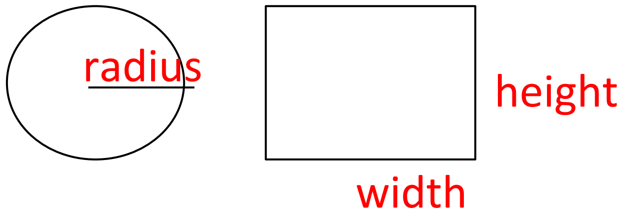
RT Embedded
Linux Solutions

Python is a high-level programming language, in other words it supports Object Oriented Programming.

The concept of Object Oriented Design development is having smart objects running the flow of your program instead of simple data-fields (several basic variables connected to a single structure) with an associated functions.

An object is a made out data (attributes) and actions that can be performed on that data (methods)

For example we could make a model out of a sqaure by making an object with width and height attributes, and with methods to calculate its area and perimater



Procedural Development

RT Embedded
Linux Solutions

<u>Procedural thinking (C):</u>

What actions do I want to perform?

✓ The main function will create the instances of the data structures defined elsewhere (my code or an existing library code)

✓ Initialize the variables and data structures

  ✓ Main will call input functions

  ✓ Validate the input of the user

  ❑ recall those functions if the input is incorrect

✓ Run my algorithm

❑--maybe I need to let the user the option to re-run my algorithm--

✓ Clean-up dynamically allocated memory upon exit

✓ Exit the program returning the relevant status

OOD Development

- Unlike procedural programming, which emphasizes algorithms, OOD emphasizes the data and code integrity.
- We design objects as if they were real life elements, and classify it’s structure using a ‘class’
- The idea is to design data objects (classes) that address the essential features of a problem.
- A class defines:
  1. Data used to represent an object
  2. Operations (code / functions / methods) that can be performed on that data.
- This approach accurately represent the issues we want our program to deal with.

OOD Development

<u>Object Oriented thinking (C++, C#, Java, python, etc ..):</u>

1. What objects do I need?

   ✓ In order to keep track of different shapes in my program, I'll design some objects to represent them (build a class for each object).

   ❑ How do I represent the object's data itself ?

      ✓ Create the internal variables within my classes

   ❑ What functionality should my objects have ?

      ✓ Create access methods to provide control over my internal variables

      ✓ Create a constructor / destructor to auto-init/clean my objects

      ✓ Create more methods to handle my solution

- Build an interface which will allow interacting with the user and my objects. I can use some APIs from my OS

Essential Features

Python provides some basic and advanced built-in data-types, and also lets you build your own objects in OOD.

Object Oriented main features implemented by python are:

- Auto-Construction and Auto-Destruction:

The creation of an object usually involves an initialization of it. It is a common practice to have your objects initialized to have a reasonable value.

You can also define more then one way to initialize your objects. Python supports function overloading ..

- Polymorphism:

Polymorphism's main Idea is to keep the concept of typical operations of different objects. For example all shapes should have the operation of printing themselves, calculating the perimeters, area etc.. It is done easily in python.

Essential Features (2)

RT Embedded Linux Solutions

- Operators overloading and Default Parameters:

  Just like in C++/C#/Java, you can define an operator functionality like add e.g +. The amount of parameters will determine which function will be called.

  Python supports special methods that refer to common operators.

  For example, you can create a Point class, and implement the addition of two point objects with ease.

- Encapsulation and data hiding:

  Encapsulation stands for the bundling of data-fields and functions into a single idea (class). Think of it as a capsule you use to get the cure.

  Hiding stands for allowing / preventing access to the object's internal components. Sometime you need to restrict the user's access to your object in order to preserve the data's integrity.

Essential Features (3)

- Inheritance:

  Inheritance is the way to establish a relationship between different object.

  This creates a relationship called is-a, which means the child object is derived from the same base-type object - the parent object. The opposite is a has-a relationship, which means that it has a component as a data-member of the new class.

C-tors, D-tors and Variables

RT Embedded Linux Solutions

```
# the following code is saved as Shapes.py
class Point:
        count = 0        # A variable shared for all Point instances.

        def __init__(self):
                Point.count = Point.count + 1     # Shared-Class variable count update

                self.x = 0        # The Object’s local variable X
                self.y = 0        # The Object’s local variable Y

        def show(self):
                print(‘x = %d, y = %d” % (self.x, self.y))

        def __del__(self):
                Point.count = Point.count – 1 # Shared-Class variable count update
```

C-tors, D-tors and Variables

RT Embedded Linux Solutions

The following code shows an example fo using the code:

```
If __name__ == '__main__':
        pt1 = Point()
        pt2 = Point()
        print('Point Count =', Point.count)
        pt1.show()   # will print x = 0, y = 0
        pt2.show()   # will print x = 0, y = 0
        del pt1, pt2
```

To use the module from another module just use:

```
from Shapes import Point
```

Polymorphism

RT Embedded Linux Solutions

```
# the following code is an update to Shapes.py
class Line:
        count = 0       # A variable shared for all Point instances.

        def __init__(self):
            Line.count = Line.count + 1    # Shared-Class variable count update

            self.p1 = Point()      # The Object's local variable p1
            self.p2 = Point()      # The Object's local variable p2

        def show(self):
            print('p1.x = %d, p1.y = %d, p2.x = %d, p2.y = %d" % (p1.x, p1.y, p2.x, p2.y))

        def __del__(self):
            Line.count = Line.count – 1 # Shared-Class variable count update
```

Polymorphism

RT Embedded
Linux Solutions

```
If __name__ == '__main__':
        pt1 = Point()
        ln1 = Line()

        print('Point Count =', Point.count)

        pt1.show()   # will print x = 0, y = 0
        ln1.show()   # will print p1.x = 0, p1.y = 0, p2.x = 0, p2.y = 0

        del pt1, ln1
```

To use the module from another module just use:

```
from Shapes import Point, Line
```

Operator Overloading

RT Embedded
Linux Solutions

```
# the following code is saved as Shapes.py
class Point:
        …
        def __add__(self, other):       # this function will be invoked when issuing p1 + p2
                return Point(self.x + other.x, self.y + other.y)

        def __sub__(self, other):       # this function will be invoked when issuing p1 - p2
                return Point(self.x - other.x, self.y - other.y)

        def __str__(self):     # this function will be invoked when casting to string (e.g. print)
                return ‘x=‘+str(self.x)+’y=‘+str(self.y)
```

For a full set of Operators, go to http://docs.python.org/3/reference/datamodel.html

Operator Overloading (2)

Method, Description & Sample Call

__init__ (self [,args...])
Constructor (with any optional arguments)
Sample Call : obj = className(args)

__del__(self)
Destructor, deletes an object
Sample Call : del obj

__repr__(self)
string representation (usually for interactive uses)
Sample Call : repr(obj)

__str__(self)

Printable string representation
Sample Call : str(obj)

__lt__ (self, x)
return true if self is "less than" x
Sample Call : obj < x

other related operators are:

__gt__	greater than	(>)
__eq__	equals	(==)
__ne__	not equals	(!=)
__ge__	greater or equals	(>=)
__le__	less or equals	(<=)

It's enough to override __lt__ __eq__ and __le__ and python will figure out the rest.

Encapsulation & Data Hiding

We can still access the internal variables of an object.

But, if we call the internal variable __myvar, it will be hidden.

```
class Point:
        count = 0       # A variable shared for all Point instances.

        def __init__(self):
                Point.count = Point.count + 1    # Shared-Class variable count update

                self.__x = 0      # The Object’s local variable X is now hidden
                self.__y = 0      # The Object’s local variable Y is now hidden

        def x(self)
                return self.__x
        def y(self)
                return self.__y
```

Inheritance

RT Embedded Linux Solutions

```
# the following code is also a continuum to Shapes.py

class Point3D(Point):
        def __init__(self):
            Point.__init__()    # we want to initialize x and y from the previous constructor.
            self.__z = 0

        # def x(self) already exists from inheritance ..
        # def y(self) already exists from inheritance ..

        def z(self):
          return __z
```

os module

Python os

OS File Navigation:

- os.getcwd()

  get the current working directory
- os.chdir(path)

  change current working directory
- os.listdir([path])

  get the list of files in the working directory
- os.mkdir(path [, mode])

  creates a directory
- os.rmdir(path)

  removes a directory

Python os (2)

RT Embedded Linux Solutions

OS Processes :

- os.getpid()

  returns the process id
- os.getuid()

  returns the user id
- os.getgid()

  returns the group id

- os.system(“shell command”)

  executes the command and returns the exit_status
- os.kill(pid, sig)

  sends a signal to a process

Sockets

Sockets

RT Embedded
Linux Solutions

When a program thread needs to connect to another local, or a remote program thread (like when using wide area network such as the Internet), it uses a software component called a socket.

The socket opens a connection for the program, allowing data to be read and written (can be over the network).

It is important to note that these sockets are software, not hardware.

There are three types of sockets:

- Local socket (Local machine)
- Inet Stream socket (TCP/IP connection)
- Inet Datagram socket (UDP/IP connection)

Socket data is in binary form

Like with open binary files, sockets send and receive data in bytes form. this means that we need a protocol for converting those bytes into useful data. for example if we want to send string data we can use the encode decode methods of str. if we want to send integers we can use the to_byte from_byet methods of int.

For more information review refresh your knowledge on what we learned from page 124 about working with binary data.

Additionally, our protocol will need some way to know when to send and when to receive. A simple design could be to make sure that each “send” from one side has a corresponding “receive” from the other.

We also need to make sure to know how much data we’re supposed to read. we can have our protocol define a limit or exact bytes to be read, or we can have some bytes values to be the indicators (delimiters) to tell us when one data ends and another begins.

Sockets Client Server

RT Embedded
Linux Solutions

These terms refer to the two applications which will be communicating with each other to exchange some information, thus creating a socket connection. One of the two apps acts as the initiative side – uses a client socket, and the other acts as the daemon side – uses a server socket.

Client socket:

This is the side which typically makes a request for information. After getting the response this process may terminate or may do some other processing

Sockets Client Server

RT Embedded
Linux Solutions

Server socket:

This is the process which takes a request from the clients. After getting a request from the client, this process will do required processing and will gather requested information and will send it to the requestor client.

Once done, it becomes ready to serve another client. Server process are always alert and ready to serve incoming requests.

Sockets API

RT Embedded
Linux Solutions

- socket(socket_family, socket_type, protocol=0):
    - socket_family: either AF_UNIX (local connection) or AF_INET (inet connection
    - socket_type: This is either SOCK_STREAM or SOCK_DGRAM
    - Protocol: not used.
- bind(): (server side)
    - This method binds address (hostname, port number pair) to socket.
- listen(): (server side)
    - This method sets up and start TCP listener.
- accept(): (server side)
    - This passively accept TCP client connection, waiting until connection arrives (blocking).
- connect(): (client side)

Sockets TCP server example

RT Embedded
Linux Solutions

```
#!/usr/bin/python    # This is server.py file

import socket        # Import socket module

s = socket.socket()  # Create a socket object

s.bind(('', 12345))  # Bind socket to the host & port (any,12345)

s.listen(5)          # Now wait for client connection.


while True:
          c, addr = s.accept()       # Establish connection with client.
          print('Got connection from', addr)
          c.send('Thank you for connecting'.encode())
          c.close()      # Close the connection
```

```
#bind((host, port)) host <- 'ip address of an interface' port <- number

#Host = '192.116.30.2' #eth0 ip for example

#Port = 2134

#s.bind((Host, Port))      # Bind socket to the host & port (any,12345)
```

Sockets TCP server example (key notes)

RT Embedded Linux Solutions

First of all, it is recommended to disable the firewall. Other option (and recommended one) is to add rules for your specific connection.

s = socket.socket()

- The default of socket() is TCP socket (AF_INET, SOCK_STREAM)

s.bind(('', 12345))

- Notice the double parenthesis: '((' and '))' those are must
- bind() expects ip / host and port., empty host means any host
- You can either give an IP or a Host/Domain, you can also translate them using socket.gethostbyname() / socket.gethostbyaddr(). For your IP/Host socket.gethostname()

s.listen(5)

- listen() expects amount of connections, 0 means no limit

Try running netstat to see the connections' creation

Sockets TCP client example

RT Embedded
Linux Solutions

```
#!/usr/bin/python          # This is server.py file

import socket              # Import socket module

s = socket.socket()        # Create a socket object

host = '192.168.0.1'             # Connect to ip address
port = 12345               # Connect to ip port


s.connect((host, port))
print(s.recv(1024).decode())
s.close                    # Close the socket when done
```

Sockets UDP server example

RT Embedded
Linux Solutions

```
#!/usr/bin/python  # This is server.py file

import socket                  # Import socket module

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)                        # Create a socket object

port = 12345                   # Connect to ip port

s.bind(('', port))      # Bind socket to the host & port (any,12345)


#all done for udp server connection

data, addr = s_srv.recvfrom(1024)
s.sendto('message'.encode(), (addr, port))

s_srv.close()                       # Close the connection
```

Sockets UDP client example

RT Embedded
Linux Solutions

```
#!/usr/bin/python     # This is server.py file

import socket         # Import socket module

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)                #
Create a socket object

port = 12345          # Connect to ip port


#all done for udp client connection

s.sendto('message'.encode(), (addr, port))

s_srv.close()                # Close the connection
```

Sockets Local-Sock server example

RT Embedded Linux Solutions

```
#!/usr/bin/python     # This is server.py file

import socket         # Import socket module

s = socket.socket(socket.AF_UNIX)       # Create a socket object

s.bind('/home/user/localsock')          # Bind to the socket file

s.listen(5)           # Now wait for client connection.


while True:
        c, addr = s.accept()            # Establish connection with client.
        print('Got connection from', addr)
        c.send('Thank you for connecting'.encode())
        c.close()                # Close the connection
```

Sockets Local-Sock client example

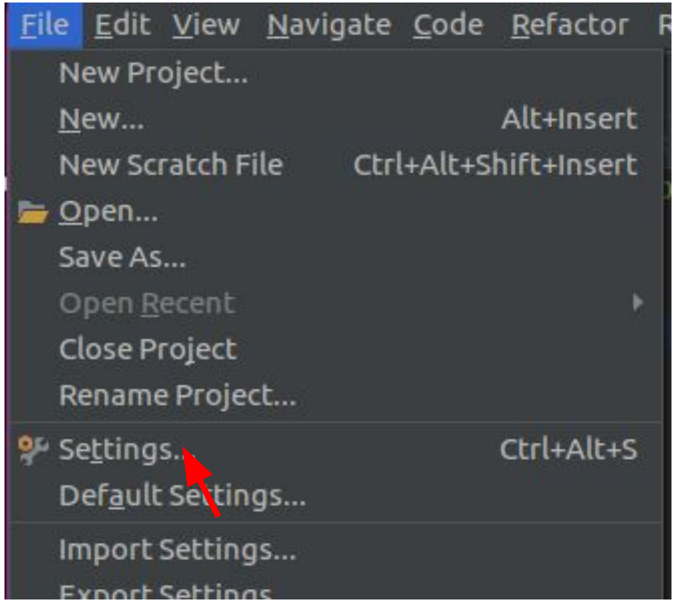
RT Embedded
Linux Solutions

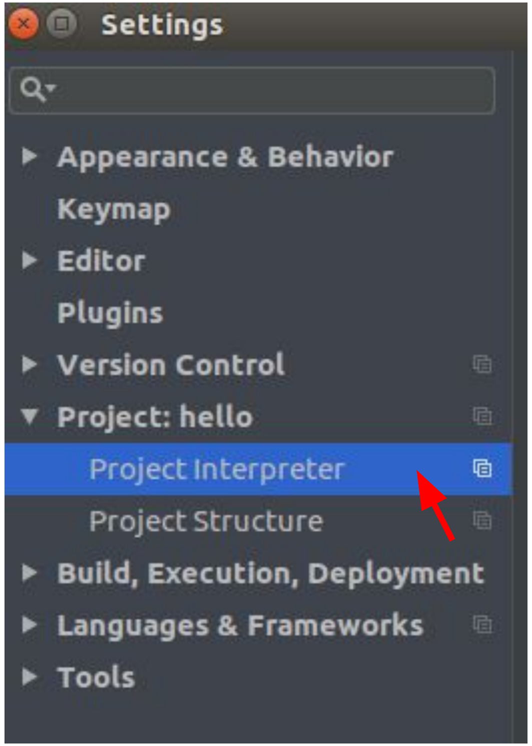
```
#!/usr/bin/python           # This is server.py file
import socket               # Import socket module
s = socket.socket(socket.AF_UNIX)      # Create a socket object
s.connect('/home/user/localsock')

print(s.recv(1024).decode())

s.close()                # Close the socket when done
```

serial

Installing new modules with pip from PyCharm

Go to File → settings → project interpreter

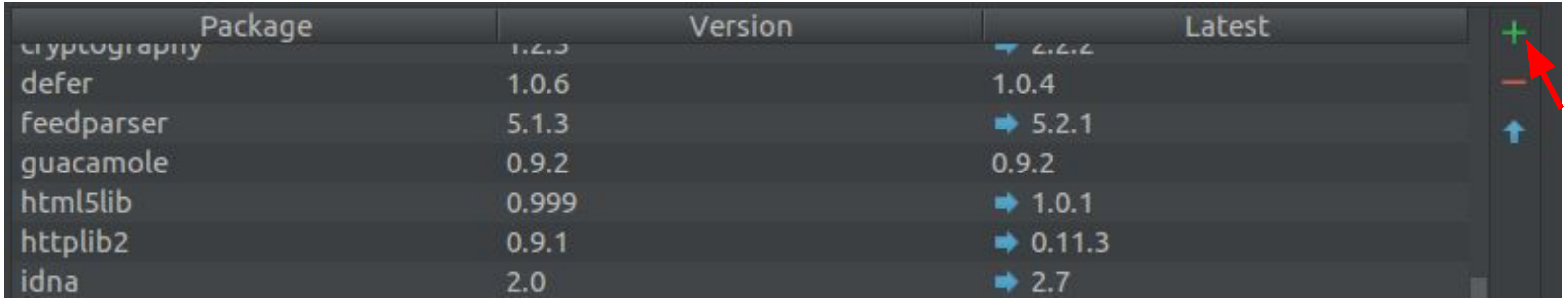




Installing new modules with pip from PyCharm

RT Embedded Linux Solutions

In the list of packages click on the green “plus” button (alternatively, you can just double click on one of the packages on the list

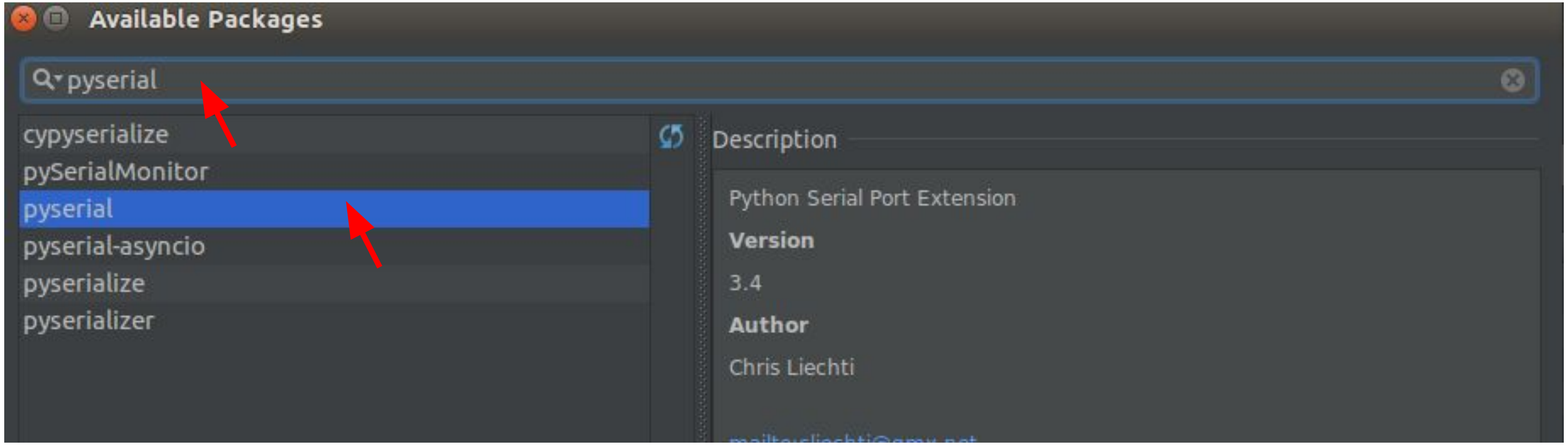


Package	Version	Latest
cryptography	1.2.3	➡ 2.2.2
defer	1.0.6	1.0.4
feedparser	5.1.3	➡ 5.2.1
guacamole	0.9.2	0.9.2
html5lib	0.999	➡ 1.0.1
httplib2	0.9.1	➡ 0.11.3
idna	2.0	➡ 2.7

Installing new modules with pip from PyCharm

RT Embedded Linux Solutions

In the “Available Packages” windows type the package you wish to install in he search bar (“pyserial” in our case). Then select the package from the list below



Python Serial Port Extension

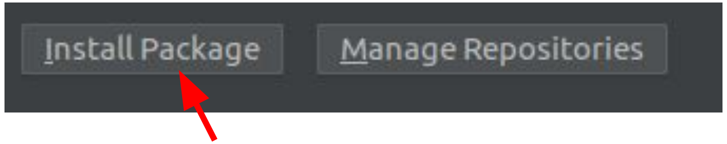
Version

3.4

Author

Chris Liechti

finally click on “install package” below. Give pyCharm some time to install and set everything up.



Serial

Python introduces a unified component for serial communication, called pySerial.

This module encapsulates the access for the serial port.

It provides backends for Python running on Windows, Linux, BSD (possibly any POSIX compliant system), Jython and IronPython (.NET and Mono).

The module to be imported is named “serial” (automatically selects the appropriate backend).

Serial (2)

RT Embedded
Linux Solutions

Those are the key features of pySeial:

-Same class based interface on all supported platforms (Linux, windows, mac..).

-Access to the port settings through Python properties.

-Support for different byte sizes, stop bits, parity and flow control with RTS/CTS and/or Xon/Xoff.

-Working with or without receive timeout.

-File like API with “read” and “write” (“readline” etc. also supported).

-The files in this package are 100% pure Python.

-The port is set up for binary transmission. No NULL byte stripping, CR-LF translation etc. (which are many times enabled for POSIX.) This makes this module universally useful.

-Compatible with io library (Python 2.6+)

-RFC 2217 client (experimental), server provided in the examples.

Serial API Overview

RT Embedded Linux Solutions

The main object used to create serial connection is the Serial class, with the serial module.

A Serial object has properties for the connection, such as the port name, baud-rate, byte size, parity bit and stop bit.

To create an instance do the following:

```
import serial

s = serial.Serial('/dev/tty/USB0')

# possible values be shown using 'print ser_obj.BAUDRATES'
s.baudrate = 9600             #

# possible values are FIVEBITS, SIXBITS, SEVENBITS, EIGHTBITS
s.byte_size = serial.EIGHTBITS

# possible values PARITY_EVEN, PARITY_MARK, PARITY_NAMES, PARITY_NONE, PARITY_ODD, PARITY_SPACE
s.parity = serial.PARITY_NONE

# possible values STOPBITS_ONE, ITS_ONE_POINT_FIVE, STOPBITS_TWO
s.stopbits = serial.STOPBITS_ONE
```

Serial API Overview

RT Embedded Linux Solutions

Another way to set Serial's properties is through initialization:

```
s = serial.Serial(port=None,
                  baudrate=9600,
                  bytesize=EIGHTBITS,
                  parity=PARITY_NONE,
                  stopbits=STOPBITS_ONE,
                  timeout=None)
```

Serial API Overview

Another important configuration for pySerial is setting the timeout parameter.

Setters and getters exists for this as well.

Possible values are as follows:

- timeout = None: wait forever
- timeout = 0: non-blocking mode (return immediately on read)
- timeout = x: set timeout to x seconds (float allowed)

For example, setting read time out to be 1 millisecond:

```
s.timeout = 0.1
```

To set write timeout to be non-blocking:

```
s.write_timeout = 0
```

Serial API Overview

RT Embedded Linux Solutions

After creation of the Serial instance it is opened by default. You can use the close method to close the connection (discards the receive buffer and any incoming receive data). You can later reopen the serial with the open method.

```
import serial

s = Serial('/dev/tty/USB0')

ser_obj.timeout =0.1

print( s.is_open ) # you can check if the connection is open with is_open property

                    #you can also set it to True to open and False to close

# Performing I/O methods such as ser_obj.read(...)

s.close()

s.open()  # re-opening the serial connection
```

Serial also supports the "with" header so you can do the following and have it close automatically:

```
with Serial('/dev/tty/USB0') as s:

        # do serial operations here
```

Serial API Overview

RT Embedded Linux Solutions

When the device is opened, reading and writing to it methods can be called. It is similar to writing / reading from a file. I/O API's are as follows:

s.read(n=1)

Reads the given amount of bytes, defaults to 1. On no-timeout, will wait until exactly n bytes are read. on timeout will read as many as available (<=n) and return that many, can also return an empty set of bytes.

s.readall()

Reads all the receive buffer.

s.readline()

Reads up until the end of the line.

s.readlines()

Reads all the receive buffer, and converts them to a list of line strings.

print(s.in_waiting)

Print how many bytes are in the receive buffer, ready to be read.

Serial API Overview

RT Embedded
Linux Solutions

```
s.write(bytes)
```

Writes the given bytes to the

```
print(s.out_waiting)
```

Printe how many bytes are in the output buffer (after write operation) waiting to be transmitted to the other side. (Don't be surprised if this always returns zero, since after a write operation the data is usually transmitted quickly and is already on the other side, before reading the value of out_writing property.

Here is a basic example to get you started:

```
with Serial('/dev/tty/USB0') as s:
    s.read() # reads one byte
    s.read(10) # reads 10 bytes
    s.write('this is my data'.encode()) # recall how to convert other types of data to bytes (Files - "Working With Bytes" section)
```

PySerial Documentation

You can find more information for working with PySerial in the following link:

General: https://pyserial.readthedocs.io/en/latest/pyserial.html

Short introduction: https://pyserial.readthedocs.io/en/latest/shortintro.html

API: https://pyserial.readthedocs.io/en/latest/pyserial_api.html

Programs for Emulating Serial Connection:

For Windows: you can install **com0com**, and use the installed "Setup" application to set a pair of connection their port name

download link:

https://sourceforge.net/projects/com0com/

For linux: you can install the **socat** command and run it with the following arguments:

```
socat -d -d PTY PTY
```

it will print a pair of open serial ports (starts with "/dev/pts/" and a number). these ports will stay open as long as socat is running (close it with CTRL-C)

SQLite in Python

Installing DB Browser

RT Embedded Linux Solutions

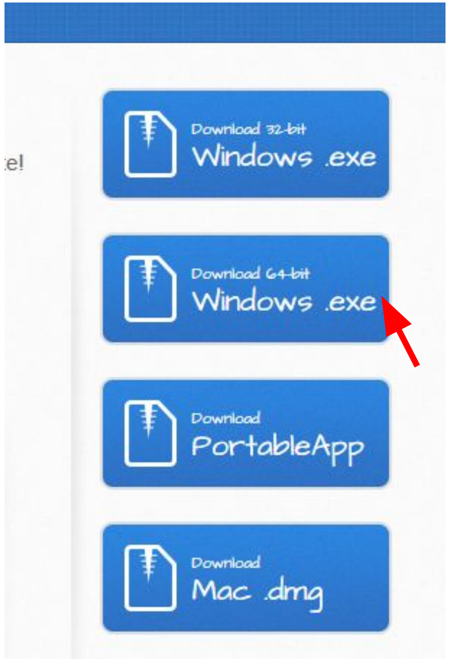
We will be using Sqlite Browser in our course for creating and working with sqlite databases.

To install on Ubuntu type in terminal: sudo apt install sqlitebrowser

To install on Windows go to this website:
https://sqlitebrowser.org/

And select a download from the available list on the right and install the file.

Usage of the program will be explained in the lectures



Python MySQLdb

RT Embedded Linux Solutions

MySQLdb is an interface for connecting to a MySQL database server from Python.

make sure you have MySQLdb installed on your machine

Open database connection

db = sqlite3.connect("filename”)

#other databases will require four arguments (“address","username","password","DBNAME")

prepare a cursor object using cursor() method

cursor = db.cursor()

execute SQL query using execute() method.

cursor.execute("SELECT VERSION()")

Python MySQLdb (2)

RT Embedded Linux Solutions

Fetch a single row using fetchone() method.

```
data = cursor.fetchone()
print("Database version :", data)
```

disconnect from server

```
db.close() #you can also use the “with” header we’ve learned about in Files!
```

Python MySQLdb (3)

RT Embedded Linux Solutions

Example: Delete

```
#!/usr/bin/python
import MySQLdb
# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
# prepare a cursor object using cursor() method
cursor = db.cursor()
# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
```

Python MySQLdb (4)

RT Embedded
Linux Solutions

```
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()
# disconnect from server
db.close()
```

Python MySQLdb (5)

RT Embedded Linux Solutions

COMMIT Operation:

Commit is the operation, which gives a green signal to database to finalize the changes, and after this operation, no change can be reverted back.

Here is a simple example to call commit method.

```
db.commit()
```

ROLLBACK Operation:

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use rollback() method.

Here is a simple example to call rollback() method.

```
db.rollback()
```

Regular Expressions

Regular Expressions

RT Embedded Linux Solutions

Regular expressions are special sequence of characters that help you match or find other strings (or sets of strings), using a specialized syntax held in a pattern. (used in UNIX)

in order to use Regular Expression you have to import the module, **re.**

import re

Instead of using \\ to write one backslash '\' as the RE string, Which makes the string difficult to be understood. We use Python's raw string notation for regular expressions, backslashes are not handled in any special way in a string literal prefixed with 'r'.

For Example:Instead of writing a regular string as a “\\s+”, we only write r”\s+”, and so on.

Compile: a method of 're' module which compiles regular expression into a pattern, for Example:

ptrn = re.compile(“[0-9]+”)

Also can accept an optional flags argument.

Regular Expressions(2)

RT Embedded Linux Solutions

Match Function:

re.match(pattern, string, flags=0)

This function matches re patterns to string with optional flags.

Where: **patterns** – the regular expression to be matched, **string** – this is the string, which would be searched to match the pattern at the begining of string, **flags** – you can specify different flags using "OR" (|). *we'll provide it to you soon...*

To get information about the matching string, *the match object also have several methods and attributes:*

group() - *returns the string matched by the re.*

start() *- returns the starting position of the match*

end() *- returns the ending position of the match*

span() *- returns a tuple containing the (start,end) positions of the match*

Regular Expressions (3)

RT Embedded
Linux Solutions

Optional Flags:

re.I - Performs case-insensitive matching.

re.L - Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B).

re.M - Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).

re.S - Makes a period (dot) match any character, including a newline.

re.U - Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.

re.X - Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

Regular Expressions (4)

RT Embedded Linux Solutions

Search function:

Search function is a function which checks for a match anywhere in the string (not only at the beginning of the string, as the match function).

re.search(pattern, string, flags=0)

Search And Replace:

This function is searching for a string, and replace all the occurrences of the RE pattern in string with the *repl*, if *max is not* provided then it will replace all the occurrences, if max provided it will replace the *max* number of substrings.

re.sub(pattern, repl, string, max=0)

Regular Expressions (5)

RT Embedded
Linux Solutions

re.findall(pattern, string, flag=0) :

find all the substrings where the re matches, and returns them as a list.

re.finditer(pattern, string, flag=0):

Finditer() method returns a sequence of match object instances as an iterator.

re.split(pattern, string, maxsplit):

Split the string into a list, splitting it wherever the RE matches.

Multithreading

Multithreading

What is Thread:

Thread is like a processes, are a mechanism to allow a program to do more than one thing at time. Thread runs concurrently, Linux kernel schedules them asynchronously, context switching them from time to time to give others a chance to execute.

The thread exist within a process, when you invoke a program, Linux creates a new process and in that process creates a single thread, which runs the program sequentially.

A program which runs only one thread called **single-threaded**, processes that contain multiple threads called **multithreaded.**

Multithreading

Start a new thread in thread module:

import _thread

thread.start_new_thread (function, args[, kwargs])

This method call enables a fast and efficient way to create new threads in both Linux and Windows. The method call returns immediately and the child thread starts and calls function with the passed list of agrs. When function returns, the thread terminates.

The **thread** module is limited. So there is another module which provides much more powerful, high-level support for thread module.

import threading

Multithreading

threading module:

It provides all the methods of the module thread and some additional methods:

threading.activeCount(): returns the number of thread objects that are active.

threading.currentThread(): returns the number of thread objects in the caller's thread control.

threading.enumerate()*:* returns a list of all thread objects that are currently active.

In addition, the module **threading** provides a **Thread** class which have the following methods:

run(), start(), join([time]), isAlive(), getName(), setName()

Multithreading

Class Thread:

run(): the entry point for the thread.

start(): starts a thread by calling the **run()** method.

join([timeout]): join method wait for threads to terminate.

isAlive(): checks whether a thread is still executing.

getName(): returns the name of the thread.

setName(): sets the name of the thread.

Multithreading

RT Embedded Linux Solutions

Create a thread using threading module:

You need to:

- Define a new Class which inherit the Thread class.
- Override the ***__init__(self [,args])*** method to add additional arguments.
- Override the ***run(self [,args])*** method to implement what the thread should do when it start to run.

Once you have created the new Thread subclass, you can create an instance of it and then start a new thread by invoking the ***start(),*** which in turn calls ***run()*** method.

Multithreading

RT Embedded Linux Solutions

Synchronizing Threads:

Python provide a simple locking mechanism that allows to synchronize threads. A new lock is created by calling ***lock()*** method.

The ***acquire()*** method is used to force threads to run synchronously, the optional blocking parameters enables you to control whether the thread waits to acquire the lock. If blocking set to 0, the thread returns immediately with a 0 value (if the lock cannot be acquired) and with 1 if the lock was acquired.

If blocking is set to 1, the thread blocks and wait for the lock to be released.

The ***release()*** method of the new lock object is used to release the lock when it is no longer required.

Multithreading

RT Embedded Linux Solutions

Multi-threading Priority Queue:

Queue module allows you to create a queue which have the following methods to control it:

- ***get()*** - remove and return an item from the Queue.
- ***put()*** - add an item to the Queue
- ***qsize()*** - returns the current number of the items
- ***empty()*** - returns True if the Queue is empty, otherwise returns False.
- ***full()*** - returns True if the Queue if Full, otherwise returns False.

RT Embedded
Linux Solutions

TK GUI with the tkinter module

GUI In Python

tkinter:

The Tkinter is a module for the standard Python interface to the Tk GUI toolkit.

make sure you have Tkinter installed on your machine.

Tkinter provides the following Widgets:

Button, canvas, comboBox, entry, frame, label, listBox, menu, menuButton, message, progressBar, radioButton, scrollBar, text and so on...

And it provides the following top-level windows:

tk_chooseColor, tk_chooseDirectory, tk_dialog, tk_getOpenFile, tk_getSaveFile, tk_messageBox, tk_popup, toplevel.

Also provides Three geometry managers:

- ***Place:*** positions widgets at absolute locations.
- ***Grid:*** arranges widgets in a grid
- ***Pack:*** packs widgets into a cavity

GUI In Python

RT Embedded Linux Solutions

Label Widget:

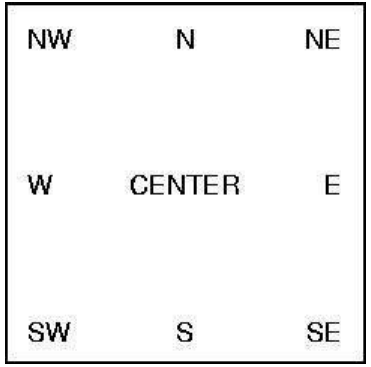
Its a display box where you can place a text or an image.

lbl= Label(master, options,....)

●Master: represents the parent window

●Options: can be more than option to control the Label itself. To add more than one option you need to separate them by commas, here is a list of some options:

- ***anchor*** (to control the position of the text in the widget if it has more space than text needs), options are: ***NW, N, NE, W, CENTER, E, SW, S, SE.***



GUI In Python

- **bg** (background), **fg** (foreground), **bd** (size of the border around)
- **bitamp**, **image** – to display a graphic image.
- ***justify*** **–** specify how multiple lines of text will be aligned, **CENTER, LEFT, RIGHT**, or you can write them as a string for example: justify=”center”**.**
- ***Cursor*** **–** change the mouse cursor to a specific pattern such as: **“arrow”, “circle”, “clock” , “cross”, “dotbox”, “exchange”, “fluer”, “heart”, “man”, “mouse”, “pirate”, “plus”, “shuttle”, “sizing”, “spider”, “spraycan”, “start”, “target”, “tcross”, “trek”, “watch”.**

GUI In Python

RT Embedded Linux Solutions

- ***Height:*** the vertical dimension of the frame
- ***text:*** to display one or more lines of text in a label widget.
- ***textvariable:*** set the label's text from a variable of class stringVar().
- ***padx:*** Extra space added to the left and right of the text within the widget. Default is 1.
- ***pady:*** Extra space added above and below the text within the widget. Default is 1.
- ***relief:*** Specifies the appearance of a decorative border around the label. The default is ***FLAT (RAISED, SUNKEN, GROOVE, RIDGE)***

GUI In Python

RT Embedded
Linux Solutions

Creating a Label with a text inside:

```
1       from tkinter import *
2       root = Tk()
3       w = Label(root, text=”First Example”)
4       w.config(bg=”red”, fg=”black”,...)
5       w.pack()
6       root.mainloop()
```

- 1: import everything from Tkinter module (Tk toolkit)
- 2: creates a Tk root widget which is a window with a little bar and some other decoration provided by the window manager. The root (master) widget has to be created before any other widget (only one root widget)

GUI In Python

- 3: contains the Label widget, first parameter is the name of the parent window (root), so the Label is a child of the root widget. Next parameter is the text which is shown on the label.
- 4: we can configure the different options by calling the config() method
- 5: the pack method tells Tk to fit the size of the window to the given text
- 6: the window won't appear until we enter the Tkinter event loop, the script will remain in the event loop until we close the window.

Labels can contain text and images, to make the label contain the image we need to set the path of the image, for example:

W = Label(root, image= “<path-to-image>”).pack(side=”left”)

GUI In Python

RT Embedded Linux Solutions

obj.config(options,...):

We can change the configure of the any object by calling the config() method and set the options we want to modify.

After we create the object, we need to place the object on the window, to do that we have to use one of the geometry managers:

- ***Pack()*** **-** This geometry manager organizes widgets in blocks before placing them in the parent widget.
- ***Grid()*** **-** This geometry manager organizes widgets in a table-like structure in the parent widget.
- ***Place()*** **-** This geometry manager organizes widgets by placing them in a specific position in the parent widget.

GUI In Python

widget.pack(pack-options) method:

This geometry manager organizes widgets in blocks before placing them in the parent widget. **Pack-Options:**

- ***Expand:*** if set ***True***, widgets expands to fill any space, if ***False*** it doesn't expand
- ***Fill:*** determines whether widget fills any extra space allocated to it by the packer, the options are:
    - ***NONE*** – default
    - ***X*** – fill only horizontally
    - ***Y*** – fill only vertically
    - ***BOTH*** – fill both horizontally and vertically
- ***Side:*** determine which side of the parent widget packs against:
    - ***TOP*** (default) , ***LEFT, RIGHT, BOTTOM***

GUI In Python

RT Embedded
Linux Solutions

widget.grid(grid_options) method:

This geometry manager organizes widgets in a table-like structure in the parent widget. Here are the **Grid-Options:**

- ***column :*** The column to put widget in; default 0 (left most column).
- ***columnspan:*** How many columns widget occupies; default 1.
- ***ipadx, ipady :***How many pixels to pad widget, horizontally and vertically, inside widget's borders.
- ***padx, pady :*** How many pixels to pad widget, horizontally and vertically, outside v's borders.
- ***row:*** The row to put widget in; default the first row that is still empty.
- ***rowspan :*** How many rowswidget occupies; default 1.

GUI In Python

widget.place(place_options) method:

This geometry manager organizes widgets by placing them in a specific position in the parent widget. Here are the **Place-Options:**

- ***anchor :*** The exact spot of widget other options refer to: may be **N, E, S, W, NE, NW, SE, or SW, default is NW**(the upper left corner of widget).
- ***bordermode :*** ***INSIDE*** (the default) to indicate that other options refer to the parent's inside (ignoring the parent's border); ***OUTSIDE*** otherwise.
- ***height, width :*** Height and width in pixels.
- ***relheight, relwidth :*** Height and width as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget. (default is 0)
- ***relx, rely :*** Horizontal and vertical offset as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget. (default is 0)
- *x, y* **:** Horizontal and vertical offset in pixels. (default is 0)

GUI In Python

RT Embedded
Linux Solutions

Within everyone of the Geometry Managers there is a method to get information about the object's properties:

- ***pack_info()***
- ***grid_info()***
- ***place_info()***

GUI In Python

RT Embedded Linux Solutions

Attributes:

- ***Dimension:*** Various lengths, widths, and other dimensions of widgets can be described in many different units.
  - If you set a dimension to an **integer**, it is assumed to be in **pixels**.
  - You can specify units by setting a dimension to a string containing a number followed by. **c (centimeters), i (inches), m (millimeters), p (printer's point - 1/72”)**
- Common Dimensions options:
  - ***Borderwidth, highlightthickness, padx,pady, selectborderwidth, wraplength, height, width,***

GUI In Python

Attributes:

- ***Colors:*** Tkinter represents colors with strings. There are two general ways to specify colors in Tkinter:
    - You can use a string specifying the proportion of red, green and blue in hexadecimal digits. For example, "#fff" is white, "#000000" is black, "#000fff000" is pure green, and "#00ffff" is pure cyan (green plus blue).
    - You can also use any locally defined standard color name. The colors **"white", "black", "red", "green", "blue", "cyan", "yellow", and "magenta"** will always be available.
- Common Colors options:
    - **background(bg),** **foreground(fg)**

GUI In Python

Attributes:

- ***Fonts:*** two ways to specify font type:
  - Tuple fonts:
    - As a **tuple** whose first element is the **font family**, followed by a **size** in points, optionally followed by a string containing one or more of the style modifiers **bold**, **italic**, **underline** and **overstrike**.
    - For Example:

("Times”, “24”. “bold”, “italic”,”underline")

or "Times 24 bold italic underline"

GUI In Python

Attributes:

- ***Fonts:***
  - Font object:
    - Import the tkFont module to create Font Object using its constructor:

    import tkFont

    font = tkFont.Font(options,...)

    - The options are:
    - **family:** The font family name as a string.
    - **size:** The font height as an integer in points.
    - **weight: "bold"** for boldface, **"normal"** for regular weight.

GUI In Python

RT Embedded
Linux Solutions

Attributes:

- The options are (continue):
- **slant: "italic"** for italic, **"roman"** for unslanted.
- **Underline: 1 (True)** for underlined text, **0 (False)** for normal.
- **Overstrike: 1 (True)** for overstruck text, **0 (False)** for normal.

GUI In Python

Text Widget:

- A text widget is used for multi-line text area.
- can be used to display links, images, and HTML, even using CSS styles.
- text widgets can also be used as simple text editors or even web browsers.

We create a text widget by using the Text() method. The Text() method returns a Text Object.

GUI In Python

RT Embedded Linux Solutions

Text Widget Methods:

- ***delete(startindex [,endindex])*** - This method deletes a specific character or a range of text.
- ***get(startindex [,endindex])*** - This method returns a specific character or a range of text.
- ***index(index)*** - Returns the absolute value of an index based on the given index.
- ***insert(index [,string]...)*** - This method inserts strings at the specified index location.
- ***see(index)*** - This method returns true if the text located at the index position is visible.

GUI In Python

RT Embedded Linux Solutions

Text Widget Options:

- ***height* –** the height of the widget in **lines** (not pixels).
- ***width* –** the width of the widget in **characters**.
- ***highlightbackground* –** the color if the widget are not focused.
- ***highlightcolor* –** the color of the widget when it is focused
- ***state* –** to set whether the text widget response to the keyboard and mouse or not, set ***NORMAL*** to let it response, for not responding widget set ***DISABLED***.
- ***xscrollcommand*** **-** To make the text widget horizontally scrollable, set this option to the **set()** method of the horizontal scrollbar.
- ***yscrollcommand* -** To make the text widget vertically scrollable, set this option to the **set()** method of the vertical scrollbar.

GUI In Python

- ***Radiobutton Widget:***
  - *This widget implements a multiple-choice button, which is a way to offer many possible selections to the user and lets user choose only one of them.*
  - *In order to implement this functionality, each group of radiobuttons must be associated to the same variable.*
  - ***Radiobuttons Methods:***
    - ***deselect() -*** *Clears (turns off) the radiobutton.*
    - ***flash() -*** *Flashes the radiobutton a few times between its active and normal colors, but leaves it the way it started.*
    - ***invoke() -*** *You can call this method to get the same actions that would occur if the user clicked on the radiobutton to change its state.*
    - ***select() -*** *Sets (turns on) the radiobutton.*

GUI In Python

Radiobutton Widget Options:

In addition to the previous widget's options, here are more options to the Radiobutton:

– ***variable*** - The control variable that this radiobutton shares with the other radiobuttons in the group. This can be either an **IntVar** or a **StringVar**.

– ***value*** – the value that is returned from radiobutton when a user choose the one of the options (sets this value to the RadioButton's control variable which is an **IntVar \ StringVar**).

– ***State*** **-** The default is state=***NORMAL***, but you can set state=***DISABLED*** to gray out the control and make it unresponsive. If the cursor is currently over the radiobutton, the state is ***ACTIVE***.

GUI In Python

CheckButton Widget:

The Checkbutton widget is used to display a number of options to a user as toggle buttons. The user can then select one or more options by clicking the button corresponding to each option.

CheckButton Methods:

- ***deselect()* -** Clears (turns off) the checkbutton.
- ***flash()* -** Flashes the checkbutton a few times between its active and normal colors, but leaves it the way it started.
- ***invoke()* -** You can call this method to get the same actions that would occur if the user clicked on the checkbutton to change its state.
- ***select()* -** Sets (turns on) the checkbutton.
- ***toggle()* -** Clears the checkbutton if set, sets it if cleared.

GUI In Python

CheckButton Widget's Options:

- *offvalue* - Normally, a checkbutton's associated control variable will be set to **0** when it is cleared (off). You can supply an alternate value for the off state by setting **offvalue** to that value.
- *onvalue* - Normally, a checkbutton's associated control variable will be set to **1** when it is set (on). You can supply an alternate value for the on state by setting **onvalue** to that value.

GUI In Python

RT Embedded
Linux Solutions

Entry Widget: w = Entry(master, option, ...)

- The Entry widget is used to accept single-line text strings from a user.
- If you want to display multiple lines of text that can be edited, then you should use the Text widget.
- If you want to display one or more lines of text that cannot be modified by the user, then you should use the Label widget.

GUI In Python

RT Embedded Linux Solutions

Entry Widget's Methods:

- ***delete (first, last=None) -*** Deletes characters from the widget, starting from first (index), up to but not including last (index). If the second argument is omitted, only the single character at position first is deleted.
- ***get() -*** Returns the entry's current text as a string.
- ***insert (index, s) -*** Inserts string **s** before the character at the given index.
- ***select_clear() -*** Clears the selection. If there isn't currently a selection, has no effect.

GUI In Python

Entry Widget's Methods:

- ***select_from (index) -*** selects character from position ***index***
- ***select_present() -*** returns ***True***, if there is a selection, else returns ***False***.
- ***select_range (start, end) -*** Selects the text starting at the ***start*** index, up to but not including the character at the ***end*** index.
- ***select_to (index) -*** Selects all the text from the ANCHOR position up to but not including the character at the given index.

Entry Widget's Options:

- ***show*** – how the characters that the user types will appear in the entry, good option to get a password by User, **show="*"**.

GUI In Python

Canvas Widget:

- Widget which let you draw 2D graphics \ plots or place widgets \ frames on a Canvas,
- Canvas objects of canvas can be:

Line, Circle, Rectangle, Image, arc, oval, polygon, window, bitamp

GUI In Python

RT Embedded
Linux Solutions

MessageBox Widget:

- The tkMessageBox module is used to display message boxes in your applications.

tkMessageBox.FunctionName(title, message [, options])

- ***FunctionName:*** *the name of the appropriate message box function.*
- ***title:*** *messagebox title.*
- ***message:*** *the text that will appear in the messagebox*
- ***options:*** *options are alternative choices that you may use to tailor a standard message box. Some of the options that you can use are default and parent. The default option is used to specify the default button, such as ABORT, RETRY, or IGNORE in the message box. The parent option is used to specify the window on top of which the message box is to be displayed.*

GUI In Python

RT Embedded
Linux Solutions

MessageBox Function:

To use these Functions you need to import the tkMessageBox module:

From tkMessageBox import *

Returns **True** \ **False** or **(OK)**

- **askokcancel()**
- ***askquestion()***
- ***askretrycancel()***
- ***askyesno()***
- ***askyesnocancel()***
- ***showerror()***
- ***showinfo()***
- ***showwarning()***

GUI In Python

Frame Widget:

- The Frame widget is very important for the process of grouping and organizing other widgets in a friendly way. It works like a container, which is responsible for arranging the position of other widgets.
- It uses rectangular areas in the screen to organize the layout and to provide padding of these widgets.

```
w = Frame ( master, option, ... )
```

GUI In Python

RT Embedded
Linux Solutions

Menu Widget:

- Create a menu (at the top of the screen).
- It is also possible to use other extended widgets to implement new types of menus, such as the OptionMenu widget, which implements a special type that generates a pop-up list of items within a selection.

w = Menu (master, option, ...)

GUI In Python

RT Embedded Linux Solutions

Menu Widget Methods:

- **add_command (options) -** Adds a menu item to the menu.
- **add_radiobutton(options) -** Creates a radio button menu item.
- **add_checkbutton(options)** - Creates a check button menu item.
- **add_cascade(options) -** Creates a new hierarchical menu by associating a given menu to a parent menu
- **add_separator()** - Adds a separator line to the menu.
- **add(type, options) -** Adds a specific type of menu item to the menu.
- **delete(startindex [, endindex])** - Deletes the menu items ranging from startindex to endindex.
- **entryconfig(index, options) -** Allows you to modify a menu item, which is identified by the index, and change its options.
- **index(item)** - Returns the index number of the given menu item label.

GUI In Python

RT Embedded Linux Solutions

Menu Widget Methods:

- **insert_separator (index) -** Insert a new separator at the position specified by index.
- **invoke (index)** - Calls the command callback associated with the choice at position index. If a checkbutton, its state is toggled between set and cleared; if a radiobutton, that choice is set.
- **type (index) -**Returns the type of the choice specified by index: either "**cascade**", "**checkbutton**", "**command**", "**radiobutton**", "**separator**", or "**tearoff**".

RT Embedded
Linux Solutions

GUI In Python

Events And Bind:

- Out application runs in mainloop() waiting for interactions (events to happen) from the user, Sometimes we want our application to get events from the keyboard (**keypress**) or from the mouse (**click**).
- Tkinter provides a mechanism to let the programmer deal with events. For each widget, it's possible to bind Python functions and methods to an event.

  widget.bind(event, handler)

- If the defined event occurs in the widget, the "handler" function is called with an event object. describing the event.

GUI In Python

RT Embedded
Linux Solutions

Events And Bind Options:

- **<Button>** - A mouse button is pressed with the mouse pointer over the widget. Left mouse button **<Button-1>**, middle button - **<Button-2>** and the right mouse button <**Button-3**>. <**Button-4**> defines the scroll up event on mice with wheel support and and <**Button-5**> the scroll down.
- <**Motion**> **-** The mouse is moved with a mouse button being held down. To specify the left, middle or right mouse button use **<B1-Motion>, <B2-Motion> and <B3-Motion>** respectively. The current position of the mouse pointer is provided in the **x** and **y** members of the event object passed to the callback, **i.e. event.x, event.y**
- **<ButtonRelease> -** to specify a button is released. For the left, middle or right mouse button use **<ButtonRelease-1>, <ButtonRelease-2>, and <ButtonRelease-3>** respectively. The current position of the mouse pointer is provided in the **x** and **y** members of the event object passed to the callback, **i.e. event.x, event.y**

GUI In Python

Events And Bind Options:

- **<Double-Button> -** To specify the left, middle or right mouse double button click use **<Double-Button-1>, <Double-Button-2>, and <Double-Button-3>**
- **<Enter>** - The mouse pointer entered the widget.
- **<Leave> -** The mouse pointer left the widget.
- **<Shift-Up> -** The user pressed the **Up arrow**, while holding the **Shift** key pressed. You can use prefixes like **Alt**, **Shift**, and **Control**.
- **<Configure> -** The size of the widget changed. The new size is provided in the width and height attributes of the event object passed to the callback. On some platforms, it can mean that the location changed.

GUI In Python

RT Embedded Linux Solutions

Events And Bind Options:

- **<FocusIn>** - Keyboard focus was moved to this widget, or to a child of this widget.
- **<FocusOut>** - Keyboard focus was moved from this widget to another widget.
- **The special keys: Cancel** (the Break key), **BackSpace**, **Tab**, **Return** (the Enter key), **Shift_L** (any Shift key), **Control_L** (any Control key), **Alt_L** (any Alt key), **Pause**, **Caps_Lock**, **Escape**, **Prior** (Page Up), **Next** (Page Down), **End**, **Home**, **Left**, **Up**, **Right**, **Down**, **Print**, **Insert**, **Delete**, **F1**, **F2**, **F3**, **F4**, **F5**, **F6**, **F7**, **F8**, **F9**, **F10**, **F11**, **F12**, **Num_Lock**, and **Scroll_Lock**.
- **<Key> -** The user pressed any key. The key is provided in the char member of the event object passed to the callback (this is an empty string for special keys).

End