

---

# **MicroPython for Satlink 3 Documentation**

*Release 1.8.4*

**Damien P. George, contributors, and Sutron Corporation**

**Jul 28, 2017**



# CONTENTS

<b>1</b>	<b>Python first time setup &amp; configuration</b>	<b>1</b>
1.1	1. Download & Install LinkComm . . . . .	1
1.2	2. Download & Install Python . . . . .	1
1.3	3. Download & Install Pyinstaller . . . . .	3
1.4	4. Download & Install PyCharm . . . . .	3
1.5	5. Testing out .py to .exe converter . . . . .	5
1.6	6. Python PyQt5 GUI . . . . .	6
1.7	7. Connect PyCharm into external programs like linkcomm or micropython . . . . .	6
1.8	8. Configure PyCharm for program development using LinkComm . . . . .	9
1.9	9. Configure PyCharm with SL3 API for auto completion . . . . .	11
1.10	10. Setting docstring stub in PyCharm . . . . .	13
<b>2</b>	<b>MicroPython libraries</b>	<b>15</b>
2.1	Python standard libraries and micro-libraries . . . . .	15
2.2	MicroPython-specific libraries . . . . .	16
2.3	Libraries specific to the Satlink 3 . . . . .	19
<b>3</b>	<b>The MicroPython language</b>	<b>39</b>
3.1	Overview of MicroPython Differences from Standard Python . . . . .	39
3.2	Code examples of how MicroPython differs from Standard Python with work-arounds . . . . .	41
3.3	The MicroPython Interactive Interpreter Mode (aka REPL) . . . . .	56
3.4	Maximising Python Speed . . . . .	60
3.5	MicroPython on Microcontrollers . . . . .	65
<b>4</b>	<b>Python and MicroPython Resources</b>	<b>71</b>
4.1	Tutorial for Xpert Basic Users . . . . .	71
4.2	Python Tutorials: . . . . .	86
4.3	MicroPython Resources: . . . . .	87
4.4	Python Videos: . . . . .	87
4.5	More Python Articles and Links . . . . .	87
4.6	MicroPython Built-In Libraries . . . . .	87
4.7	MicroPython Standard Libraries . . . . .	89
4.8	MicroPython Cross Compiler and Windows Version. . . . .	89
4.9	Udacity Free Courses . . . . .	89
<b>5</b>	<b>MicroPython license information</b>	<b>91</b>
<b>6</b>	<b>Indices and tables</b>	<b>93</b>
	<b>Python Module Index</b>	<b>95</b>



## PYTHON FIRST TIME SETUP & CONFIGURATION

These are the instructions for installing and configuring software to provide the best script development environment for the Sutron Satlink 3

### 1.1 1. Download & Install LinkComm

LinkComm is the program used to communicate with the Sutron Satlink 3 and update the firmware. It also downloads Python scripts, configures the setup to invoke them, and it can test them as well.

Below is a link to to the Sutron Beta site to download the latest version.

<http://www.sutron.com/linkcomm-beta-software/>

### 1.2 2. Download & Install Python

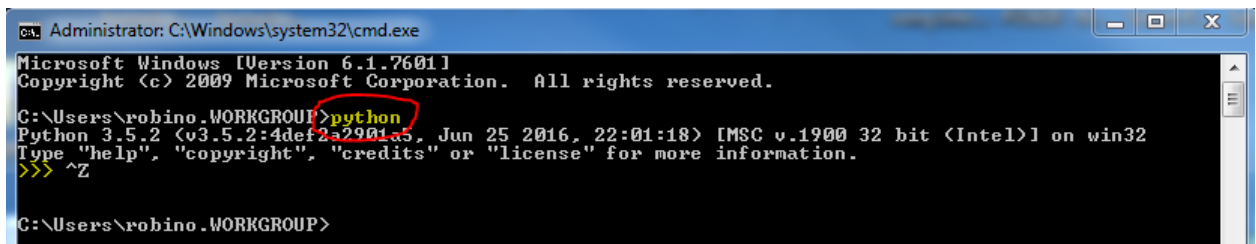
If you want to use pyinstaller which converts a python script into a standalone executable that can be run on a machine without python, you will need to download Python version 3.5.2 which is the latest version that is support by pyinstaller. Below is a link to the download.

<https://www.python.org/ftp/python/3.5.2/python-3.5.2.exe>

**a.** Run the installer and make sure to check the “Add Python 3.5 to PATH”. Do an install now rather than customize since we want pip installed in addition to python.



b. Verify installation is good and OS environment is setup by opening a command window (CMD) and typing in python and enter. You should see 3 arrow prompt show up indicating python is running. You can press Ctrl+Z to exit out of the python prompt.

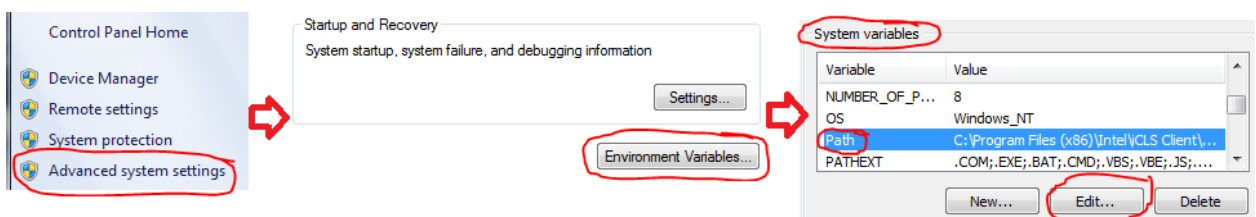


c. Installation and Environment setup is complete.

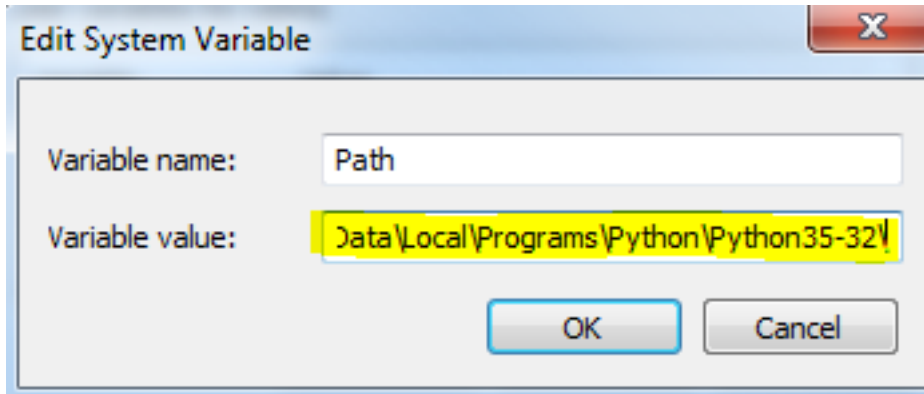
If python in CMD created an error, it most likely means you didn't click the "Add Python 3.5 to PATH" when you installed it and you need to setup your OS environment manually. A quick solution is to uninstall python and reinstall it making sure to click that check mark.

If you need to setup your environment manually, navigate to the environment variables on your system and add the path to the install location of python.exe. Only edit the environment manually if you know what you are doing otherwise you can screw up other things on your system.

Windows 7



You can append to the variable value field by putting in a “;” and pasting the path to python.exe install location on your system.



### 1.3 3. Download & Install Pyinstaller

This is not required to create python scripts. It is only needed if you want to convert your scripts into stand-alone exe files for distribution.

Open CMD window and type “pip install pyinstaller” and press enter. pip will automatically download and install pyinstaller. You may need to alter the command to “py -m pip install pyinstaller”. This was necessary on my windows 10 machine.

If everything worked as it should, you should see a line indicating successfully installed and be presented with a command prompt again. You are done with pyinstaller and it is ready for use.

### 1.4 4. Download & Install PyCharm

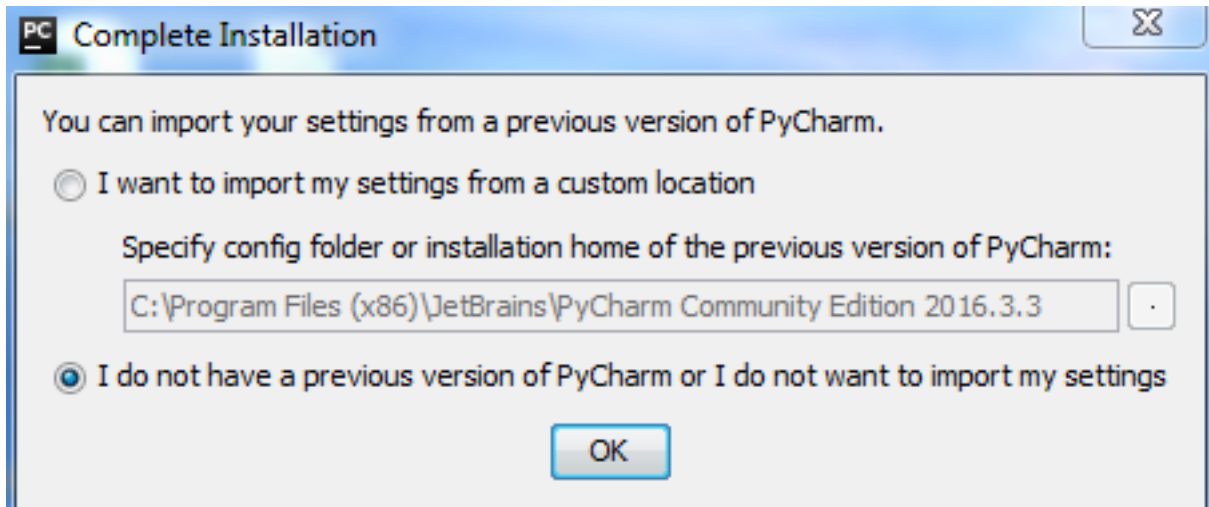
This is a python development environment that makes it easier to code in python. This is not necessary as python comes with a built-in editor called IDLE. However, it is recommend since PyCharm makes it easier to write python code, debug, and run your code.

You can download the required file from: <https://www.jetbrains.com/pycharm/download/#section=windows>

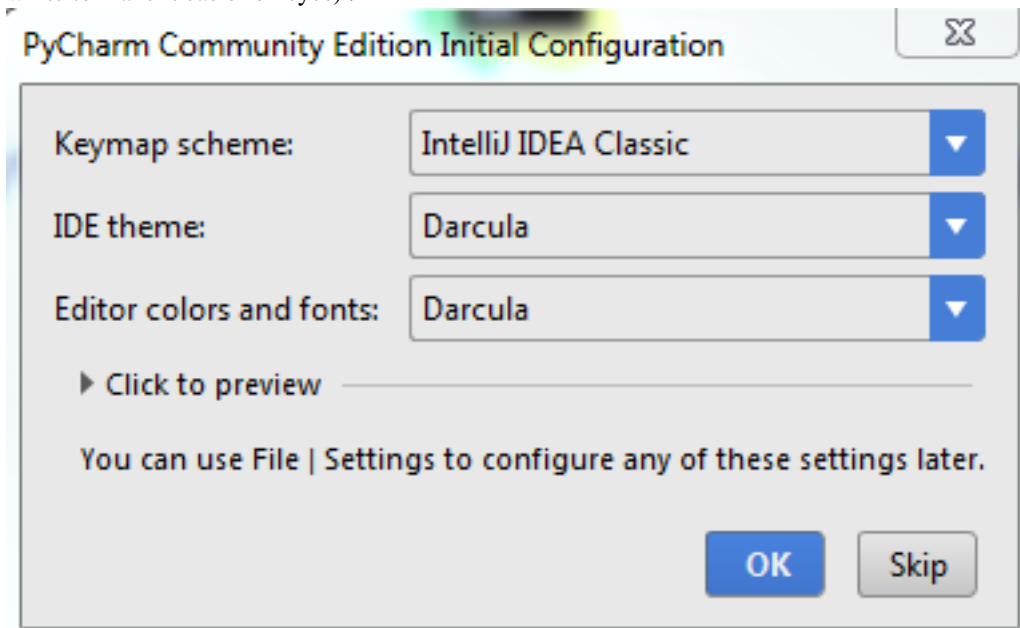
Make sure to get the Free community version so you don’t have to deal with licenses.

#### First time install & setup process:

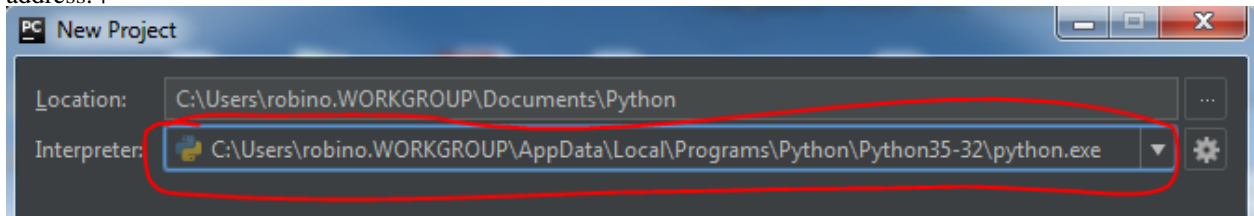
1. Run the file you downloaded and select the second option “I do not have a previous version...” |



2. Accept default configuration (you can change to Darcula theme so you have a grey background rather than white to make it easier on eyes) |



3. Create new project
4. Set default project directory for all projects.
5. For the interpreter, select the drop down and it should automatically detect python is installed and give the address. |

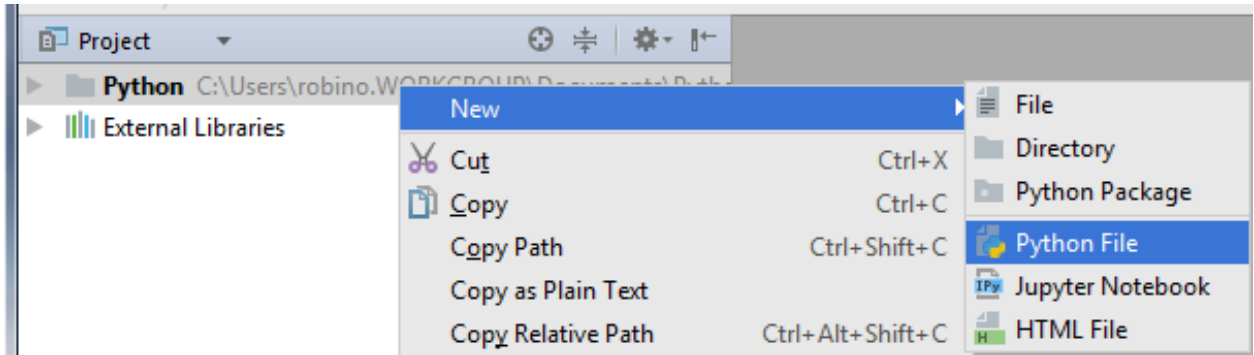


6. Installation and setup is complete.

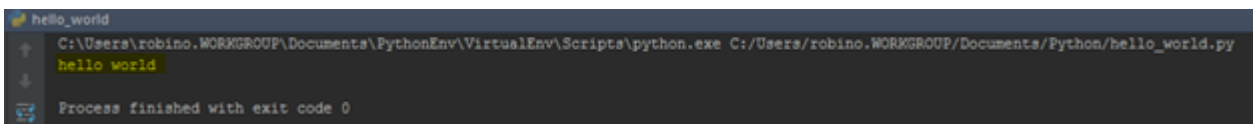
### Creating new Python file



1. Create a new python file by right clicking python > new > Python file.



2. Give file name “hello\_world”
3. Type in “print(‘hello world’)”
4. Pres Ctrl+Shift+F10 (this runs hello\_world.py)
5. You should see a window at the bottom that prints out “hellow world” like below.

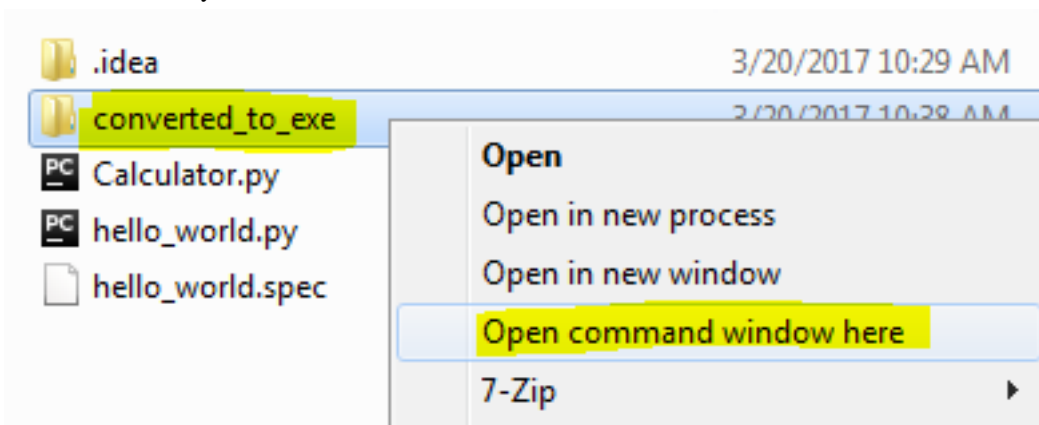


6. Done.

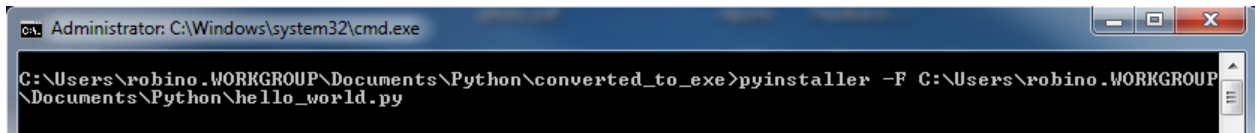
## 1.5 5. Testing out .py to .exe converter

We will create a stand-alone exe file for the hello\_world.py you just created above.

1. Navigate to the folder that contains your hello\_world.py
2. shift right click on folder and select “Open command window here”. In the below example, I created a folder within the directory to hold all the converted exe files.



3. Type in the command window “pyinstaller -F” and drag the hello\_world.py file into the command window to fill in the address. It should look something like below. The “-F” tells the installer to create a single exe file that contains everything to run the program.



```
C:\Users\robino.WORKGROUP\Documents\Python\converted_to_exe>pyinstaller -F C:\Users\robino.WORKGROUP\Documents\Python\hello_world.py
```

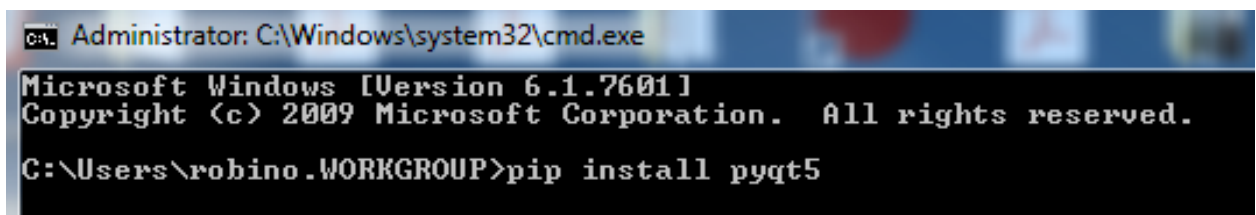
4. Your exe file is located in subfolder called “dist” (short for distribute-able). Note that if you run the exe file, a cmd window opens, prints out “hello world” and closes right away. This is because it is a very simple program with no interaction so it is done right away. You can put in a sleep statement after printing text in your program to slow this process down so you can see it.

## 1.6 6. Python PyQt5 GUI

This is also not required but recommended for those wanting to create a nice graphical user interface. I would recommend downloading PyQt5 as it is more modern and in the same family as what Linkcomm was created with. As such, Mike Nelson is a great resource with respect to Qt GUI implementation.

to install PyQt5:

1. Open CMD window and type “pip install pyqt5” enter



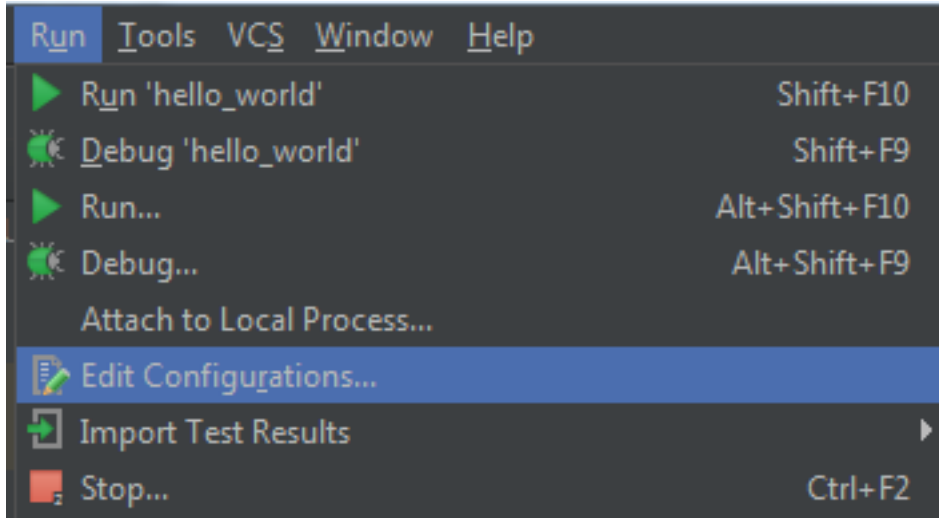
```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\robino.WORKGROUP>pip install pyqt5
```

2. Done.

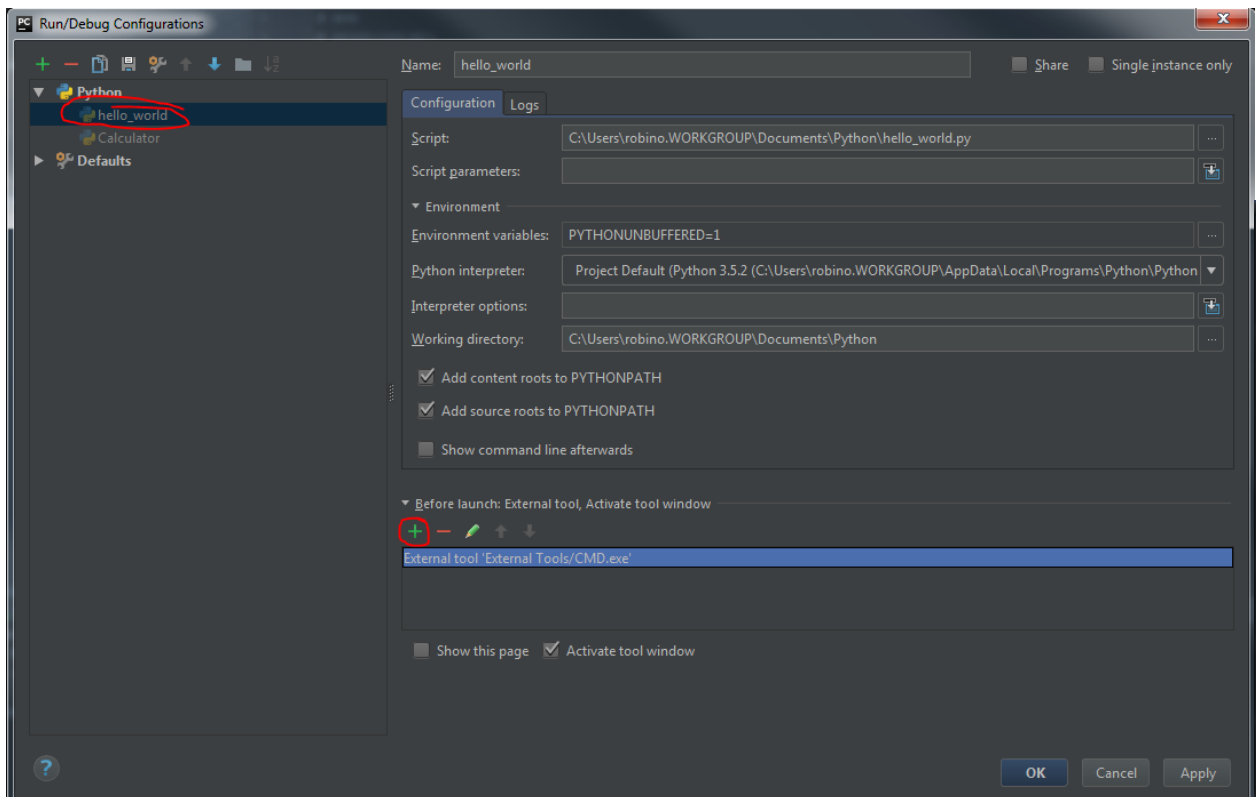
## 1.7 7. Connect PyCharm into external programs like linkcomm or micropython

PyCharm can run python programs directly in the IDE. If instead you want to send the program to an external application like Linkcomm or micropython and read any messages back into the IDE, you’ll need to setup External Tools. Below is an example of setting up an external tool. You can have several setup and assign them keyboard short cuts.

1. From main menu in PyCharm, select run and then click “Edit Configurations”

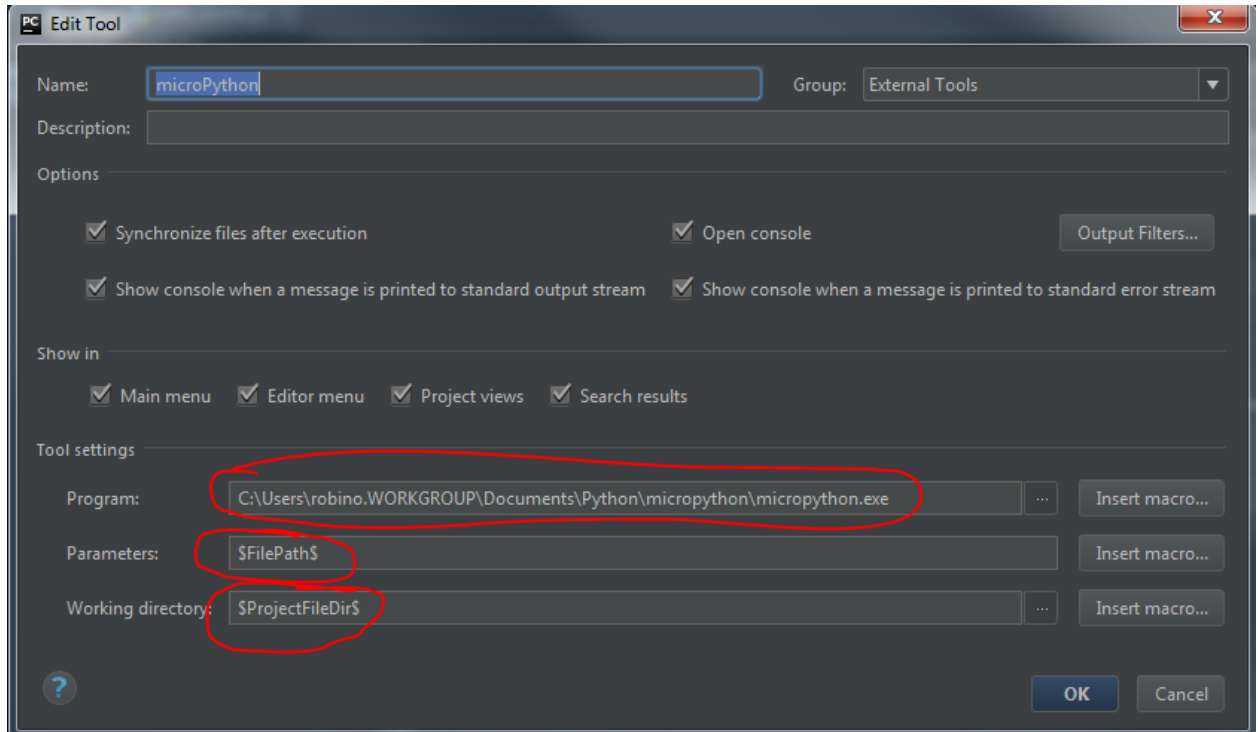


2. On the configuration screen, select the specific script you want to apply the external tool to or make it default for all scripts. Then click the little plus sign at the bottom to add an external tool. On the external tool screen, click plus sign again to add a tool. If you already created one (example shows 3 already created), click the one you want and click ok to accept.

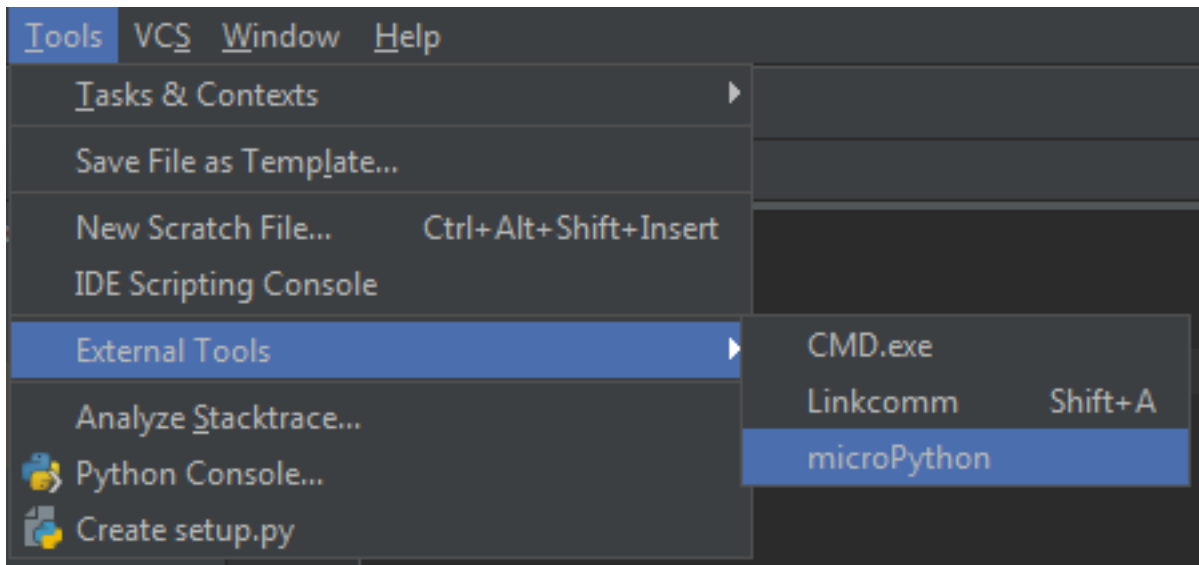


3. If you add a new external tool, you will get the screen below.
  - (a) You have to specify the path to the application you want to run. In the example, it is micropython.exe
  - (b) Set the parameter you want to send. This can be text command or in this case, I'm sending the path to my current script for micropython to run.
  - (c) Set the working directory to current project directory.

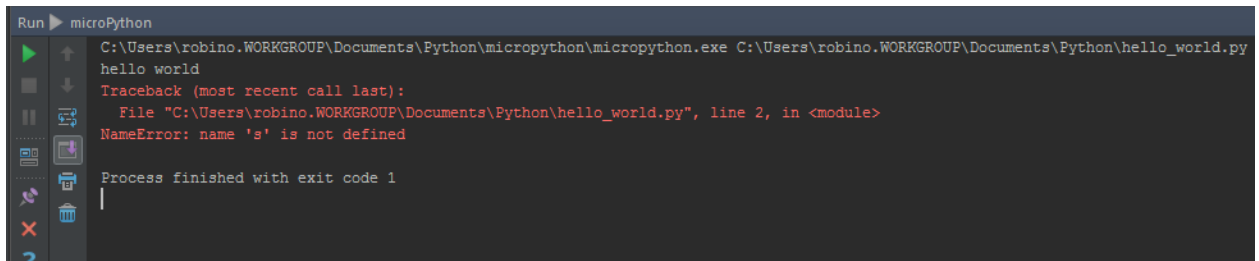
- (d) If you want to try and parse the returned message to create hyper links to error lines in the code, you can click “output Filters” and define one.



- 4. Click ok on all the menus and back out to the editor screen. You can run your new tool for the current project by going to main menu > tools > External Tools > microPython



- 5. A window will pop up at the bottom of the screen that shows the application you invoked, the parameters you sent (file path in this case), and the resulting message from the application. The example below shows that a “NameError” occurred on line 2 in my code but the first line print statement “hello World” showed up just fine.

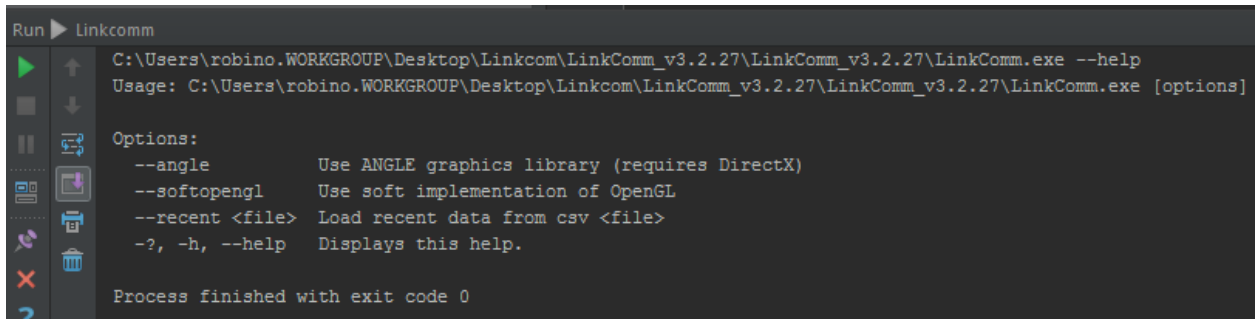


```

Run ▶ microPython
C:\Users\robino.WORKGROUP\Documents\Python\micropython\micropython.exe C:\Users\robino.WORKGROUP\Documents\Python\hello_world.py
hello world
Traceback (most recent call last):
  File "C:\Users\robino.WORKGROUP\Documents\Python\hello_world.py", line 2, in <module>
    NameError: name 's' is not defined
Process finished with exit code 1

```

- You can test linkcomm out by doing the same thing as all the setups above but for parameter to send, type in “-help”. Linkcomm returns help messages in this window. Sending a path to the python file simply opens up linkcomm but nothing happens at this time with the python file.



```

Run ▶ Linkcomm
C:\Users\robino.WORKGROUP\Desktop\Linkcom\LinkComm_v3.2.27\LinkComm_v3.2.27\LinkComm.exe --help
Usage: C:\Users\robino.WORKGROUP\Desktop\Linkcom\LinkComm_v3.2.27\LinkComm_v3.2.27\LinkComm.exe [options]

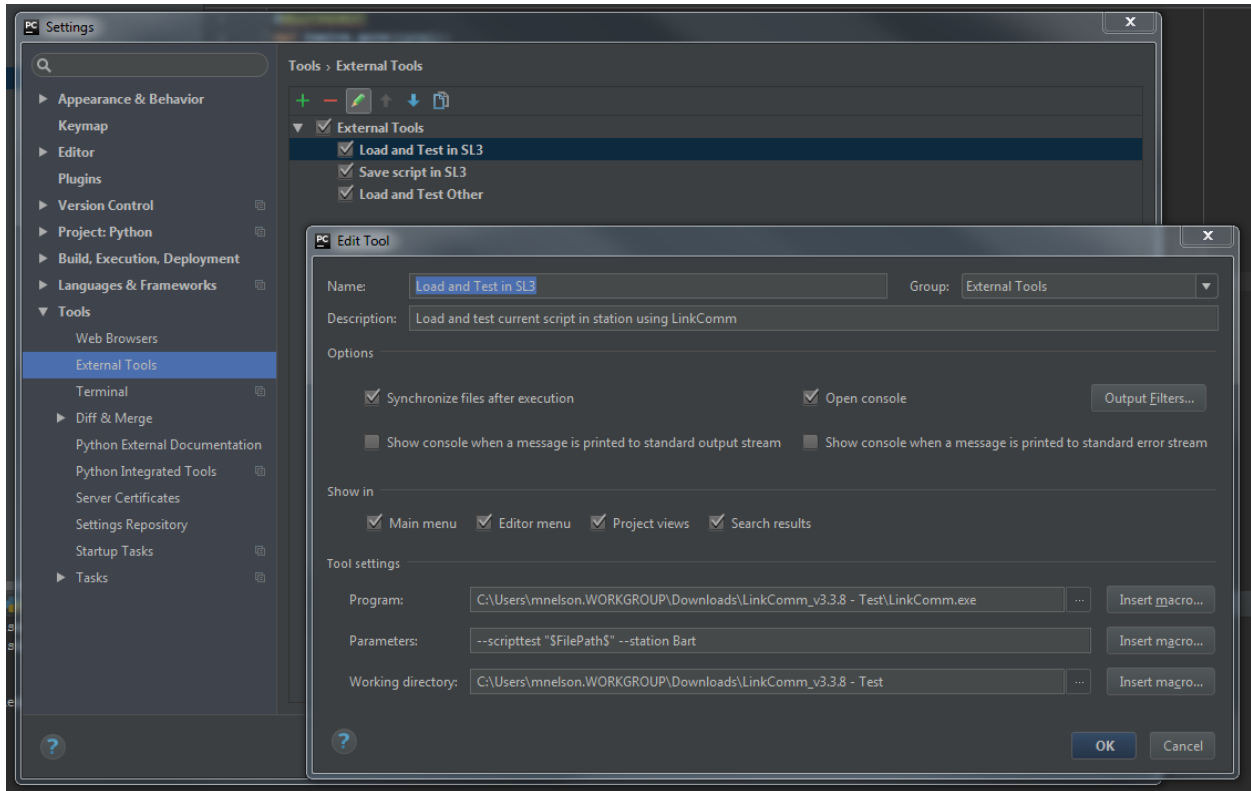
Options:
--angle           Use ANGLE graphics library (requires DirectX)
--softopengl     Use soft implementation of OpenGL
--recent <file>  Load recent data from csv <file>
-?, -h, --help   Displays this help.
Process finished with exit code 0

```

## 1.8 8. Configure PyCharm for program development using LinkComm

JIRA issue [LNK-703](#) - Getting issue details... STATUS added this functionality.

In PyCharm, add a new external tool named, “Load and Test in SL3” using the menu item, File - Settings - External Tools. Click the “+” sign to add a new external tool:



The key points to this dialog are:

1. Set “Program” and “Working directory” to point to the instance of LinkComm to run
2. Set “Parameters” to specify how you want LinkComm to connect, and what you want LinkComm to do (more on this in Configure Parameters, below)
3. Define an output filter to tell PyCharm how to identify file and line links in the output (more on this in Configure Output Filter, below)

Note: If LinkComm is already running and connected to the target station, then the request is handled by the instance already running, rather than starting a new instance that fails to connect.

### 1.8.1 8.1 Configure Parameters

The following Python-related command line options are supported by LinkComm:

Option	Description
--scripttest <file>	Load and test script with the given file path
--scriptsave	Command station to save current script
--station <name>	Connect to station <name>
--wifi	Connect using Wi-Fi
--usb	Connect to first found USB device
--usbid <device-id>	Connect to specified USB <device-id>
--serial <port>	Connect using serial <port>, e.g., COM1, COM2, etc.

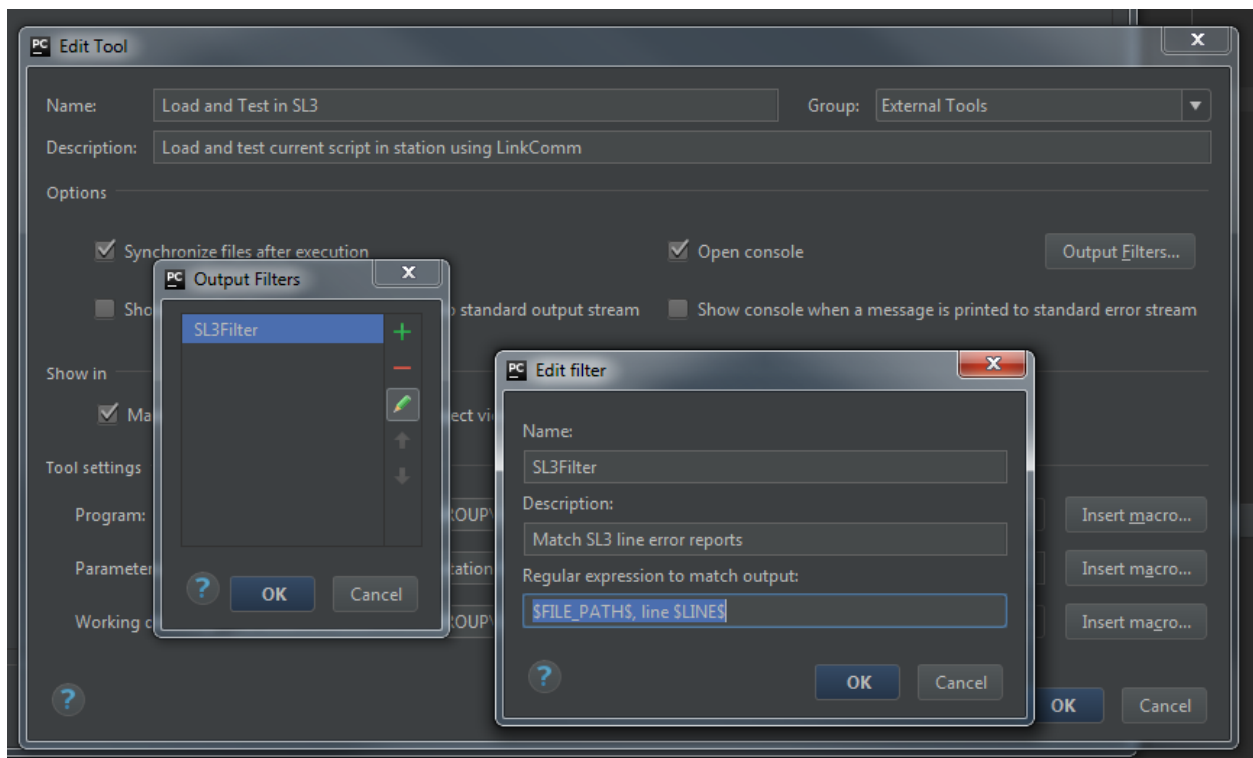
Either “--scripttest <file>” or “--scriptsave” must be specified, telling LinkComm what action to take.

Example usage in PyCharm:

Example	Description
<code>-scripttest "\$FilePath\$"</code>	Load and test "\$FilePath\$" in "New Station", using current connection settings
<code>-scriptsave</code>	Save script in "New Station", using current connection settings
<code>-scripttest "\$FilePath\$" -station EC1200</code>	Load and test "\$FilePath\$" in station named "EC1200", using current connection settings
<code>-scripttest "\$FilePath\$" -wifi</code>	Load and test "\$FilePath\$" in "New Station", using Wi-Fi connection
<code>-scripttest "\$FilePath\$" -usb</code>	Load and test "\$FilePath\$" in "New Station", using first found USB device
<code>-scripttest "\$FilePath\$" -station "EC1200" -wifi</code>	Load and test "\$FilePath\$" in station named "EC1200", using Wi-Fi connection (updates station's connection settings to Wi-Fi)

## 1.8.2 8.2 Configure Output Filter

Create an output filter in PyCharm to tell PyCharm how to find file-line links in the station's test output.

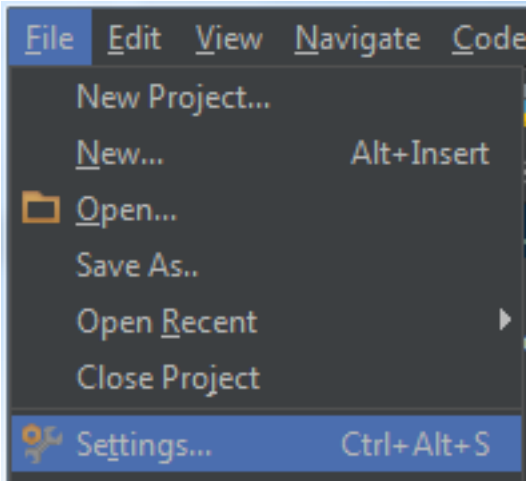


Note the regular expression used, "\$FILE\_PATH\$, line \$LINE\$". This tells PyCharm how to find file - line links in the output.

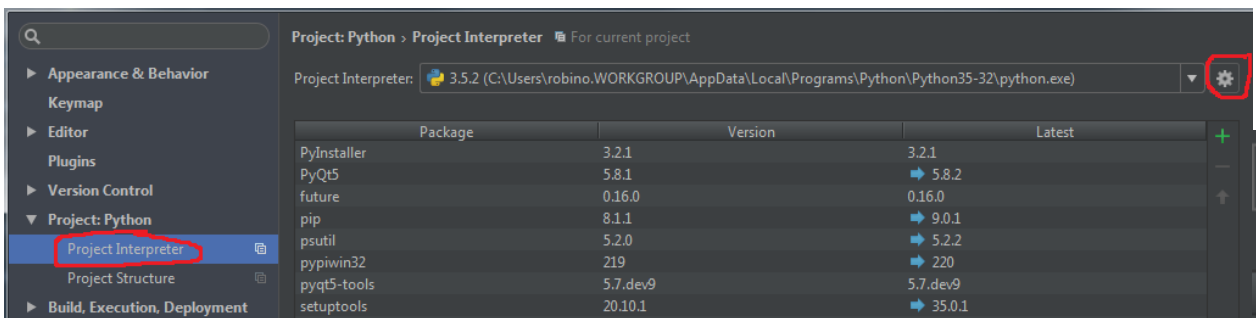
## 1.9 9. Configure PyCharm with SL3 API for auto completion

This process will allow you to import sl3 API functions written in python into your own script and have auto-completion in Pycharm on all sl3 API functions. With dummy return values in the sl3 API, you should be able to run your script directly in pycharm and quickly debug code before even loading it into sl3. SL3 API python file will be supplied by sutron and will not require any kind of installation.

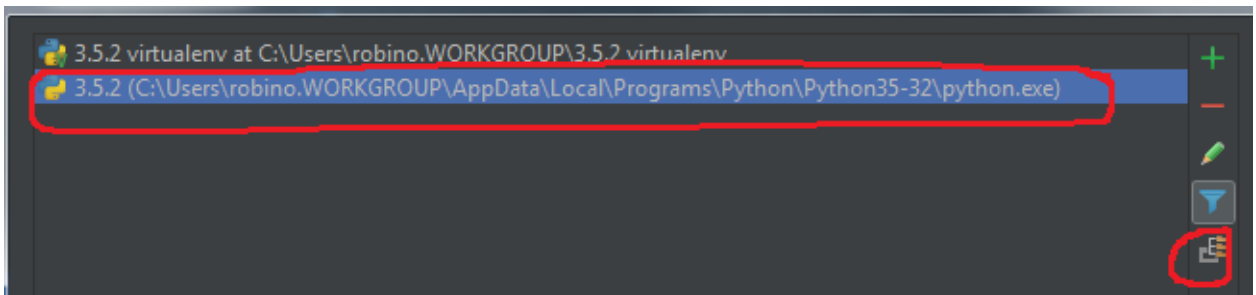
1. Click on file > settings.



2. Select Project Interpreter and then click the little Wheel on top right and select “more...”.

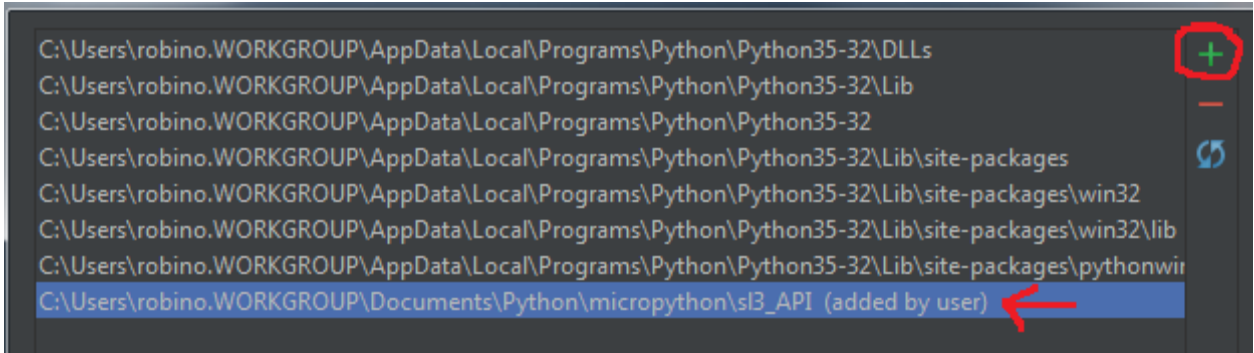


3. Select your python interpreter and click on the bottom symbol on the right side.



4. This screen shows the interpreter paths and we want to add the path to your sl3 API python file. Do this by clicking the green plus symbol top right and select the folder containing your sl3.py. It will show up as “added by user” in the list. Click ok and everything is setup. You will now need to add this statement “from sl3 import \* ” without the quotes in your script. You can now start typing any function, class or variable and pycharm will autocomplete. The sl3.py file can be edited at any time and your script will have access to everything in it without reloading or installing anything.





## 1.10 10. Setting docstring stub in PyCharm

When you define a function and immediately follow it with triple quote marks, this will automatically generate a docstring stub. See example below:

```
def dew_point(at, rh):
    """

    :param at:
    :param rh:
    :return:
    """
```

And from there you can expand on the docstring to explain the function you are creating:

```
def dew_point(at, rh):
    """
    Compute the dew point (the temperature at which water vapor will
    condense in to water)

    :param at: air temperature in deg C
    :param rh: relative humidity in %
    :return: the current dewpoint temperature in deg C
    """
```



## MICROPYTHON LIBRARIES

**Warning:** Important summary of this section

- MicroPython implements a subset of Python functionality for each module.
- To ease extensibility, MicroPython versions of standard Python modules usually have `u` (micro) prefix.
- Any particular MicroPython variant or port may miss any feature/function described in this general documentation, due to resource constraints.

This chapter describes modules (function and class libraries) which are built into MicroPython. There are a few categories of modules:

- Modules which implement a subset of standard Python functionality and are not intended to be extended by the user.
- Modules which implement a subset of Python functionality, with a provision for extension by the user (via Python code).
- Modules which implement MicroPython extensions to the Python standard libraries.
- Modules specific to a particular port and thus not portable.

Note about the availability of modules and their contents: This documentation in general aspires to describe all modules and functions/classes which are implemented in MicroPython. However, MicroPython is highly configurable, and each port to a particular board/embedded system makes available only a subset of MicroPython libraries. For officially supported ports, there is an effort to either filter out non-applicable items, or mark individual descriptions with “Availability:” clauses describing which ports provide a given feature. With that in mind, please still be warned that some functions/classes in a module (or even the entire module) described in this documentation may be unavailable in a particular build of MicroPython on a particular board. The best place to find general information of the availability/non-availability of a particular feature is the “General Information” section which contains information pertaining to a specific port.

Beyond the built-in libraries described in this documentation, many more modules from the Python standard library, as well as further MicroPython extensions to it, can be found in the [micropython-lib repository](#).

### 2.1 Python standard libraries and micro-libraries

The following standard Python libraries have been “micro-ified” to fit in with the philosophy of MicroPython. They provide the core functionality of that module and are intended to be a drop-in replacement for the standard Python library.

## 2.2 MicroPython-specific libraries

Functionality specific to the MicroPython implementation is available in the following libraries.

### 2.2.1 `machine` — functions related to the board

The `machine` module contains specific functions related to the board.

#### Miscellaneous functions

`machine.info([verbose])`

Print information about the machine including unique id, interned string usage, and memory usage. If the `verbose` argument is given then extra information is printed including a heap dump.

`machine.unique_id()`

Returns a byte string with a unique identifier of a board/SoC. It will vary from a board/SoC instance to another, if underlying hardware allows. Length varies by hardware (so use substring of a full value if you expect a short ID). In some MicroPython ports, ID corresponds to the network MAC address.

### 2.2.2 `micropython` – access and control MicroPython internals

#### Functions

### 2.2.3 `uctypes` – access binary data in a structured way

This module implements “foreign data interface” for MicroPython. The idea behind it is similar to CPython’s `ctypes` modules, but the actual API is different, streamlined and optimized for small size. The basic idea of the module is to define data structure layout with about the same power as the C language allows, and the access it using familiar dot-syntax to reference sub-fields.

#### See also:

**Module `struct`** Standard Python way to access binary data structures (doesn’t scale well to large and complex structures).

#### Defining structure layout

Structure layout is defined by a “descriptor” - a Python dictionary which encodes field names as keys and other properties required to access them as associated values. Currently, `uctypes` requires explicit specification of offsets for each field. Offset are given in bytes from a structure start.

Following are encoding examples for various field types:

- Scalar types:

```
"field_name": uctypes.UINT32 | 0
```

in other words, value is scalar type identifier ORed with field offset (in bytes) from the start of the structure.

- Recursive structures:

```
"sub": (2, {
    "b0": ctypes.UINT8 | 0,
    "b1": ctypes.UINT8 | 1,
})
```

i.e. value is a 2-tuple, first element of which is offset, and second is a structure descriptor dictionary (note: offsets in recursive descriptors are relative to a structure it defines).

- Arrays of primitive types:

```
"arr": (ctypes.ARRAY | 0, ctypes.UINT8 | 2),
```

i.e. value is a 2-tuple, first element of which is ARRAY flag ORed with offset, and second is scalar element type ORed number of elements in array.

- Arrays of aggregate types:

```
"arr2": (ctypes.ARRAY | 0, 2, {"b": ctypes.UINT8 | 0}),
```

i.e. value is a 3-tuple, first element of which is ARRAY flag ORed with offset, second is a number of elements in array, and third is descriptor of element type.

- Pointer to a primitive type:

```
"ptr": (ctypes.PTR | 0, ctypes.UINT8),
```

i.e. value is a 2-tuple, first element of which is PTR flag ORed with offset, and second is scalar element type.

- Pointer to an aggregate type:

```
"ptr2": (ctypes.PTR | 0, {"b": ctypes.UINT8 | 0}),
```

i.e. value is a 2-tuple, first element of which is PTR flag ORed with offset, second is descriptor of type pointed to.

- Bitfields:

```
"bitf0": ctypes.BFUINT16 | 0 | 0 << ctypes.BF_POS | 8 << ctypes.BF_LEN,
```

i.e. value is type of scalar value containing given bitfield (typenamees are similar to scalar types, but prefixes with “BF”), ORed with offset for scalar value containing the bitfield, and further ORed with values for bit offset and bit length of the bitfield within scalar value, shifted by BF\_POS and BF\_LEN positions, respectively. Bitfield position is counted from the least significant bit, and is the number of right-most bit of a field (in other words, it’s a number of bits a scalar needs to be shifted right to extra the bitfield).

In the example above, first UIN16 value will be extracted at offset 0 (this detail may be important when accessing hardware registers, where particular access size and alignment are required), and then bitfield whose rightmost bit is least-significant bit of this UIN16, and length is 8 bits, will be extracted - effectively, this will access least-significant byte of UIN16.

Note that bitfield operations are independent of target byte endianness, in particular, example above will access least-significant byte of UIN16 in both little- and big-endian structures. But it depends on the least significant bit being numbered 0. Some targets may use different numbering in their native ABI, but `ctypes` always uses normalized numbering described above.

## Module contents

**class** `uctypes.struct` (*addr, descriptor, layout\_type=NATIVE*)

Instantiate a “foreign data structure” object based on structure address in memory, descriptor (encoded as a dictionary), and layout type (see below).

`uctypes.LITTLE_ENDIAN`

Layout type for a little-endian packed structure. (Packed means that every field occupies exactly as many bytes as defined in the descriptor, i.e. the alignment is 1).

`uctypes.BIG_ENDIAN`

Layout type for a big-endian packed structure.

`uctypes.NATIVE`

Layout type for a native structure - with data endianness and alignment conforming to the ABI of the system on which MicroPython runs.

`uctypes.sizeof` (*struct*)

Return size of data structure in bytes. Argument can be either structure class or specific instantiated structure object (or its aggregate field).

`uctypes.addressof` (*obj*)

Return address of an object. Argument should be bytes, bytearray or other object supporting buffer protocol (and address of this buffer is what actually returned).

`uctypes.bytes_at` (*addr, size*)

Capture memory at the given address and size as bytes object. As bytes object is immutable, memory is actually duplicated and copied into bytes object, so if memory contents change later, created object retains original value.

`uctypes bytearray_at` (*addr, size*)

Capture memory at the given address and size as bytearray object. Unlike `bytes_at()` function above, memory is captured by reference, so it can be both written too, and you will access current value at the given memory address.

## Structure descriptors and instantiating structure objects

Given a structure descriptor dictionary and its layout type, you can instantiate a specific structure instance at a given memory address using `uctypes.struct()` constructor. Memory address usually comes from following sources:

- Predefined address, when accessing hardware registers on a baremetal system. Lookup these addresses in datasheet for a particular MCU/SoC.
- As a return value from a call to some FFI (Foreign Function Interface) function.
- From `uctypes.addressof()`, when you want to pass arguments to an FFI function, or alternatively, to access some data for I/O (for example, data read from a file or network socket).

## Structure objects

Structure objects allow accessing individual fields using standard dot notation: `my_struct.substruct1.field1`. If a field is of scalar type, getting it will produce a primitive value (Python integer or float) corresponding to the value contained in a field. A scalar field can also be assigned to.

If a field is an array, its individual elements can be accessed with the standard subscript operator `[]` - both read and assigned to.

If a field is a pointer, it can be dereferenced using `[0]` syntax (corresponding to C `*` operator, though `[0]` works in C too). Subscripting a pointer with other integer values but 0 are supported too, with the same semantics as in C.

Summing up, accessing structure fields generally follows C syntax, except for pointer dereference, when you need to use `[0]` operator instead of `*`.

## Limitations

Accessing non-scalar fields leads to allocation of intermediate objects to represent them. This means that special care should be taken to layout a structure which needs to be accessed when memory allocation is disabled (e.g. from an interrupt). The recommendations are:

- Avoid nested structures. For example, instead of `mcu_registers.peripheral_a.register1`, define separate layout descriptors for each peripheral, to be accessed as `peripheral_a.register1`.
- Avoid other non-scalar data, like array. For example, instead of `peripheral_a.register[0]` use `peripheral_a.register0`.

Note that these recommendations will lead to decreased readability and conciseness of layouts, so they should be used only if the need to access structure fields without allocation is anticipated (it's even possible to define 2 parallel layouts - one for normal usage, and a restricted one to use when memory allocation is prohibited).

## 2.3 Libraries specific to the Satlink 3

The following libraries are specific to the Satlink 3:

### 2.3.1 `serial` — Serial port class

The `serial` module is designed as a subset of the PySerial library with some differences due to `serial` being implemented as a built-in streaming class rather than as Python code, and some functionality not being present and/or simplified.

RS-485 is supported but rather than using a class to represent the settings, direct attributes can be configured.

**class** `serial.Serial` (*port=None, baudrate=0, bytesize=8, parity='N', stopbits=1, rtscts=False, dsrdtr=False, xonxoff=False, timeout=None, inter\_byte\_timeout=0*)

Serial objects can be created and initialised using:

```
# open RS232 at 9600 baud
port = serial.Serial("RS232", 9600)
port.close()

# create port, then defer the open until we configure:
port = serial.Serial()
port.port = "RS232"
port.baudrate = 9600
port.parity = 'N'
port.stopbits = 1
port.open()
```

`bytesize`, `parity`, and `stopbits` can be configured but do not NOT currently do anything

A `Serial` object acts like a stream object and reading and writing is done using the standard stream methods:

```
port.read(10)      # read 10 characters, returns a bytes object
port.readall()    # read all available characters
port.readline()   # read a line
port.readinto(buf) # read and store into the given buffer
port.write('abc') # write the 3 characters
```

Individual characters can be read/written using:

```
port.readchar()    # read 1 character and returns it as an integer
port.writechar(42) # write 1 character
```

To check if there is anything to be read, use:

```
port.in_waiting    # returns the number of bytrs in the input buffer
```

`__init__` (*port=None, baudrate=0, bytesize=8, parity='N', stopbits=1, rtscts=False, dsrdtr=False, xonxoff=False, timeout=None, inter\_byte\_timeout=0*)

Create a serial port object and open it

If no parameters are supplied then the port will not be opened until minimally a port is configured and `open()` is called.

#### Parameters

- **port** (*str*) – is the serial port to manage and open (ex: 'SERIAL', 'CELL').
- **baudrate** (*int*) – The clock rate (when 0 the current rate will be used).
- **bytesize** (*int*) – The number of bits per byte. (\*)
- **parity** (*str*) – The parity, 'E', 'N', or 'O'. (\*)
- **stopbits** (*int*) – The number of stop bits: 1 or 2. (\*)
- **rtscts** (*bool*) – Enable H/W flow control.
- **dsrdtr** (*bool*) – Require DTR to be asserted.
- **xonxoff** (*bool*) – Enable S/W flow control. (\*)
- **timeout** (*float*) – The time in seconds to wait for the first character (None is the max timeout of 60sec).
- **inter\_byte\_timeout** (*float*) – The time in seconds to wait between characters (None is the max, 0 means do not wait for new data).

**Returns** a Serial port object

**Return type** *Serial*

---

**Note:** (\*) These parameters may be configured, but are not supported at this time.

---

**close()**

Close port immediately

**flush()**

Wait until all serial data has been sent.

**ioctl** (*param*)

Used internally to help make serial ports compatible with sockets

**open()**

Open a Serial port with the current settings. For instance, here's an example that will open the 'RS232' port and communicate with DTR low:



```

from serial import Serial
s = Serial()
s.port = 'RS232'
s.baudrate = 115200
s.timeout = 0.5
s.dtr = 0
s.open()
s.write('AT\r')
print(s.read())

```

**Raises `SerialException`** – The method will raise `SerialException` if the port is already open, or port is invalid or unassigned.

**`read(size=256)`**

Read `size` bytes from the serial port. If a timeout is set it may return less characters as requested. With no timeout it will block until the requested number of bytes is read.

---

**Note:** One important difference when porting code which uses PySerial is that the `read()` method will return up to 256 bytes by default, to get the same behaviour as PySerial's `read()` method call `read(1)`. Something else to consider is that when working with binary data, CPython code may convert the data from bytes to a str using the “latin1” character set, but that does not exist in MicroPython; instead use `sl3.bytes_to_str()`

---

Example:

```

import serial
with serial.Serial("RS232", 9600) as port:
    port.timeout=2 #maximum time to wait before closing port
    port.inter_byte_timeout=.01 #maximum time between bytes. This makes_
↪reading port data more responsive as you do not have to wait the timeout_
↪period.
    port.reset_input_buffer() #clears buffer before reading to remove any_
↪residual data.
    buf = port.read(512)
print(buf)

```

**Parameters `size` (int)** – Number of bytes to read.

**Returns** Bytes read from port.

**Return type** *bytes*

**`readall()`**

Read as much data as possible. Similiar to `read()` but is not limited to 256 bytes.

**Returns** Bytes read from port.

**Return type** *bytes*

**`readchar()`**

Receive a single character from the port

**Returns** The character read, as an ASCII integer. Returns -1 on timeout.

**Return type** *int*

**readinto** (*buf*, *nbytes*)

Read bytes into the *buf*. If *nbytes* is specified then read at most that many bytes. Otherwise, read at most `len(buf)` bytes. Reading in to a preallocated buffer avoids the overhead of allocating new buffers on each read.

Example 1:

```
# Create a bytearray and read 5 bytes in to it.
buf = bytearray(256)
# read up to 5 bytes in to buf ( do not wait )
serial_port.timeout = 0
n1 = serial_port.readinto(buf, 5)
```

Example 2:

```
# Add a dash to the buf and read 10 more bytes after that making use
# of memoryview to reference the buf without making a copy of it.
buf[n1] = chr('-')
n1 += 1
# read 10 more bytes after the dash
mv = memoryview(buf)
# read in to a slice of the memory view (no data is copied)
n2 = serial_port.readinto(mv[n1:n1+10])
# when done, we can convert from bytearray to bytes
buf_as_bytes = bytes(buf)
```

**Parameters**

- **buf** (`bytearray`) – A bytearray to read bytes in to.
- **nbytes** (`int`) – May optionally specify how many bytes or leave out to to fill the buffer.

**Returns** Number of bytes read.

**Return type** *int*

**readline** ()

Read a line ending in the newline character ('`\n`').

**Returns** Bytes read from port.

**Return type** *bytes*

**reset\_input\_buffer** ()

Flush input buffer, discarding all its contents.

**reset\_output\_buffer** ()

Clear output buffer, aborting the current output and discarding the output buffer.

**send\_break** (*duration=0.25*)

Send break condition. Transmit line returns to idle state after given *duration*.

**Parameters** **duration** (`float`) – time in seconds to activate the break condition

**write** (*data*)

Write data to the serial port.

**Parameters** **data** (`str`) – Data to send (accepts bytes, str, bytearray, memoryview).

**Returns** Number of bytes written or None if the operation failed.

**Return type** *int*

**writechar** (*char*)

Write a single character to the port.

**Parameters** **char** (*int*) – ASCII value of a character to write.

**baudrate = 9600**

Get or set the current baudrate.

**break\_condition = 0**

When set to True activate BREAK condition, else disable. Controls TXD. When active, no transmitting is possible.

**bytesize = 8**

Does not currently do anything.

**cd = False**

Returns as a bool the state of the CD line.

**cts = False**

Returns as a bool the state of the CTS line.

**delay\_after\_tx = 0.0**

Delay in seconds before clearing RTS after an RS-485 transmission ends.

**delay\_before\_tx = 0.0**

Delay in seconds before setting RTS before an RS-485 transmission ends.

**dsr = False**

Returns as a bool the state of the DSR line.

**dsrdtr = False**

When set True, read/write operations will fail if DSR is not asserted (True).

**dtr = False**

Set DTR line to specified state (True for On, False for Off). If the value is set before opening the serial port, then the value will be applied upon open().

**in\_waiting = 0**

Get the number of bytes in the input buffer.

**inter\_byte\_timeout = None**

Sets the timeout in seconds (floating point allowed) when reading data after the initial byte is read. A value of None will cause this value to be ignored.

**is\_open = False**

Returns True if the serial port has been opened.

**loopback = False**

When set to True transmitted data over RS-485 is also received.

**name = None**

Returns the name of the selected serial port.

**out\_waiting = 0**

Get the number of bytes in the output buffer (a value of 1 indicates that at least one byte needs to be sent).

**parity = 'N'**

Does not currently do anything.

**port = None**

Get or set the serial port.

Available ports include 'RS232', 'RS485', 'CELL'.

**Warning:** Opening a port can interfere with any standard operations the Satlink 3 is performing with that port.

**ri = False**

Returns as a bool the state of the RING line.

**rs485 = False**

Enables RS-485 mode on the port. On standard serial ports, RS-485 mode will assert RTS when sending data according to the RS-485 settings.

**rts = False**

Set RTS line to specified state (True for On, False for Off). If the value is set before opening the serial port, then the value will be applied upon open().

**rts\_level\_for\_rx = True**

Determines the RTS level after an RS-485 transmission.

**rts\_level\_for\_tx = True**

Determines the RTS level during an RS-485 transmission.

**rtscts = False**

Enable hardware (RTS/CTS) flow control.

**stopbits = 1**

Does not currently do anything.

**timeout = None**

Sets the timeout in seconds (floating point allowed) when reading data. If an inter\_byte\_timeout has been set, then timeout will only apply to the first byte read. A timeout of None would traditionally wait “forever” but we limit that to no more than 60 second.

**write\_timeout = None**

Not currently implemented.

**xonxoff = False**

Not currently implemented.

## 2.3.2 s13 — Satlink 3 API

This module provides the Application Programming Interface for the Sutron Satlink. It consists of functions that provide access to Satlink’s systems.

### exception s13.SetupError

### class s13.Log (count=None, match=None, start=None, stop=None)

The Log class is used to access the log. It may be used to search the log, matching a log entry label as well as a start and a stop time.

To access any of the log methods you must create an instance of the Log class, but because the current position in the log is tracked automatically you do not have to keep a copy of the instance.:

```
>>> # For instance, creating a new Log() instance each time and accessing the
↳first 3 records in the log:
>>>
>>> Log().get_least_recent()
04/04/2017,18:01:43,RS232 Connected,0,,G
>>> Log().get_next()
04/04/2017,18:01:43,Reset Soft,278,,G
```

```

>>> Log().get_next()
04/04/2017,18:01:44,Usb Drive Connected,0,,G
>>>
>>> # is the same as creating a single instance of Log() and re-using it:
>>>
>>> l = Log()
>>> l.get_least_recent()
04/04/2017,18:01:43,RS232 Connected,0,,G
>>> l.get_next()
04/04/2017,18:01:43,Reset Soft,278,,G
>>> l.get_next()
04/04/2017,18:01:44,Usb Drive Connected,0,,G

```

Log can be used with iterators and it's very handy to pass a count to specify how many log entries you wish to process:

```

# print out the next 10 stage entries
for i in Log(10, "stage"):
    print(i)

```

Notes:

- most Log functions return an instance of the Reading class
- the current position in the log is tracked on a per thread basis to prevent conflicts.
- it's recommended to avoid creating a Log() instance in a global variable to avoid potential conflict with other threads.

**Matching by label and time** If you want to match certain labels, you can pass it in as the match property; a start date or a stop date can be specified as well.:

```

>>> # retrieve the most recent Soft Reset event
>>>
>>> Log(match='Reset Soft').get_most_recent()
05/09/2017,15:06:54,Reset Soft,115,,G
>>>
>>> # retrieve the last Soft Reset event that occurred yesterday:
>>>
>>> t = localtime()
>>> # localtime() returns the current time in a tuple which is not-modifiable,
↳ but we can convert it
>>> # into a list and create a unique copy with the list() function:
>>> t1 = list(t)
>>> # subtract 1 from the day, and set the HH:MM:SS to 00:00:00
>>> t1[2:6] = (t1[2] - 1, 00, 00, 00)
>>> # use the list() function to make another unique copy of the current time:
>>> t2 = list(t)
>>> # subtract 1 from the day, and set the HH:MM:SS to 23:59:59
>>> t2[2:6] = (t2[2] - 1, 23, 59, 59)
>>> # create a Log() instance with the label we wish to match and the start_
↳time and stop time
>>> # and call get_most_recent() to start searching for a match moving_
↳backwards in time:
>>> Log(match='Reset Soft', start=mktime(t1), stop=mktime(t2)).get_most_
↳recent()
05/08/2017,22:21:16,Reset Soft,115,,G
>>>
>>> # retrieve the next earlier matching Soft Reset event:

```

```

>>>
>>> Log(match='Reset Soft', start=mktime(t1), stop=mktime(t2)).get_prev()
05/08/2017,22:19:58,Reset Soft,114,,G
>>>
>>> # or to be more efficient we can copy the Log() instance into a variable,
↳and then just re-use it:
>>>
>>> log = Log(match='Reset Soft', start=mktime(t1), stop=mktime(t2))
>>> log.get_prev()
5/08/2017,22:11:54,Reset Soft,113,,G
>>> log.get_prev()
05/08/2017,17:59:44,Reset Soft,112,,G

```

**Iterating with the Log Class** If we want to manipulate a block of Log entries in Python, we can take advantage of the fact that the Log() class supports iteration. That means we can loop on it. Here's an example based on the previous, that displays all the soft resets that happened yesterday:

```

>>> # reset the Log to the most recent:
>>> Log().get_most_recent()
05/09/2017,22:43:54,Time After Change,4,ms,G
>>> # now loop on entries which match our criteria:
>>> for reading in log:
...     print(reading)
05/08/2017,22:21:16,Reset Soft,115,,G
05/08/2017,22:19:58,Reset Soft,114,,G
05/08/2017,22:11:54,Reset Soft,113,,G
05/08/2017,17:59:44,Reset Soft,112,,G
05/08/2017,16:12:37,Reset Soft,111,,G

```

**\_\_init\_\_** (count=None, match=None, start=None, stop=None)  
Creates an instance of the log class.

#### Parameters

- **count** – limit the number of log entries read
- **match** – label value to match
- **start** – match entries greater than or equal to this time
- **stop** – match entries less or equal to this time

**get\_least\_recent** ()  
Returns the oldest entry in the log

**Return type** *Reading*

**get\_most\_recent** ()  
Returns the newest entry in the log

**Return type** *Reading*

**get\_next** ()  
Returns the next newer entry in the log

**Return type** *Reading*

**get\_prev** ()  
Returns the previous older entry in the log

**Return type** *Reading*

**next ()**

Returns either the next newer or previous older entry depending on whichever direction was retrieved last. By default, the most recent entries that have been added to the log since the script started running is returned.

**Return type** *Reading*

**count = None**

number of Readings to iterate

**limit = 1000**

a limit on the number entries to search when looking for a match

**match = None**

a label to match

**start = None**

a start time to match

**stop = None**

a stop time to match

**class** `s13.Reading` (*time=0, label="", value=0.0, units="", quality='B', right\_digits=2, etype='M'*)

**This class is used to represent a log entry and a measurement result.**

- It is used when reading and writing to the log.
- It is also used when getting the results of a measurement reading.

**Notes**

- When a Reading is printed, the full contents are displayed using csv format.
- Two Reading's may be compared for equality (or inequality):

```
r1 = Reading(value=4.3, quality='G', label='test', time=time())
r2 = Reading(value=4.3, quality='G', label='test', time=r1.time)
r1 == r2 # evaluates to True
r1 != r2 # evaluates to False
```

**\_\_init\_\_** (*time=0, label="", value=0.0, units="", quality='B', right\_digits=2, etype='M'*)

A Reading can be initialized with another Reading or a tuple, or fields. Here are some examples of different ways a Reading can be constructed:

```
# Create a Reading by filling out the fields one by one:
r = Reading()
r.time = time()
r.value = -9.7
r.label = "at"
r.quality = 'G'
r.right_digits = 1
r.units = 'C'

# Create a Reading by passing one or more of time, label, value, units,
↳ quality, right_digits, and entry type:
r = Reading(time(), "stage", 5.2, "ft", "G")

# Create a Reading by passing keyword arguments (order doesn't matter):
r = Reading(value=4.3, quality='G', label='test', time=time())

# Create a Reading by passing another reading:
```

```
Reading(r)

# Create a Reading by passing a tuple containing the properties
Reading((time(), "stage", 5.2, "ft", "G", 2, 'M'))
```

**asctime()**

Converts the time of the Reading into an ASCII string :return: the time in MM/DD/YYYY,HH:MM:SS  
:rtype: str

**write\_log(reading)**

Writes the reading to the log:

```
reading = Reading(value=4.3, quality='G', label='test', time=time())
reading.write_log()
```

**etype = 'M'**

**Represents the type of reading**

- 'M': a measurement
- 'E': an event
- 'D': a debug message

**label = ""**

A name for the reading; a string up to 11 bytes

**quality = 'B'**

A single byte indicating the quality of the reading, with B indicating a bad value, and G indicating a good value

**right\_digits = 0**

The number of right digits of the value to show

**time = 0**

Time is represented in seconds since 1970 as returned by `utime.time()`

**units = ""**

“The units expressed as a string up to three bytes long

**value = 0.0**

The value of the reading, a float

**class s13.Serial\_number**

Provides the serial number of SL3, microboard, and transmitter board. Returned values are ASCII strings containing the serial number.

**classmethod microboard()**

**classmethod s13()**

**classmethod transmitter()**

**s13.ascii\_time(time\_t)**

Converts the provided time into an ASCII string:

```
>>> ascii_time(utime.localtime())
'07/03/2017,21:40:57'
```

**Parameters time\_t** – time as a number of seconds since 1970, just like `utime.time()`



**Returns** the time in MM/DD/YYYY,HH:MM:SS

**Return type** *str*

`s13.ascii_time_hms(time_t)`

Converts the provided time into an ASCII string HH:MM:SS:

```
>>> ascii_time_hms(utime.localtime())
'21:40:57'
```

**Parameters** `time_t` – time as a number of seconds since 1970, just like `utime.time()`

**Returns** the time in HH:MM:SS

**Return type** *str*

`s13.batt()`

Measures Satlink's supply voltage :return: battery voltage :rtype: float

`s13.bin6(num, count=3, right_digits=0, is_signed=True)`

Converts a number in to a `count` byte long 6-bit packed binary string (used for GOES formatting). The maximum number of bytes is 3, which allows a representation range of [-131072 .. 131071]. The more `right_digits` you request, the smaller the range. For instance with 2 `right_digits`, the the maximum range becomes limited to [-1310.72 .. 1310.71], and even less with fewer bytes.

Examples:

```
>>> import s13
>>> # Convert 12345 to 6-bit pseudo-binary:
>>> s13.bin6(12345)
b'c@Y'
>>> # Convert BV in to a 3-byte 18-bit value:
>>> BV = 12.32
>>> s13.bin6(BV, 3, 2)
b'@SP'
```

**Parameters**

- **num** (float) – A number to convert to pseudo-binary format.
- **count** (int) – The number of bytes to format (1 to 3).
- **right\_digits** (int) – The number of decimal digits to retain via scaling.
- **is\_signed** (bool) – True if negative values should be allowed.

**Returns** A byte string containing 6-bit binary data.

**Return type** *bytes*

`s13.bin_to_str(num, count)`

Converts a number in to a (`count`) byte long binary string. For example, `bin(65,1)` would return `b'A'`. If the number is a floating point number, then `count` must be either 4 or 8 to have the number stored in IEEE 32 bit or 64 bit single or double precisions respectively. If the `count` is not 4 or 8 (or -4 or -8), then the number is treated as an integer. The sign of `count` determines the byte order. When `count` is positive, the most significant byte is stored first (Compatible with ARGOS 8-bit binary formats). When `count` is negative, the least significant byte is stored first.

Example:

```
>>> import s13
>>> s13.bin_to_str(1094861636, 4)
b'ABCD'
```

**Parameters**

- **num** (*int, float*) – Integer or float number to be converted to byte string
- **count** (*int*) – Number of bytes to converted num to. Byte count must be large enough to hold the number passed in.

**Returns** Returns binary string representation of the number passed in.

**Return type** *bytes*

**s13.bit\_convert** (*string, type*)

Converts a binary string of specific format tye into a number. This is the opposite of bin function. The following types are supported:

Type	string contents
1	4 byte integer, least significant byte (LSB) first
2	4 byte integer, most significant byte (MSB) first
3	IEEE 754 32 bit float, least significant byte (LSB) first
4	IEEE 754 32 bit float, most significant byte (MSB) first
5	IEEE 754 64 bit float, least significant byte (LSB) first
6	IEEE 754 64 bit float, most significant byte (MSB) first

Example:

```
>>> import s13
>>> s13.bit_convert(b'DCBA', 1)
1094861636
>>> s13.bit_convert(b'@\x9c\xcc\xcd', 4)
4.9
```

**Parameters**

- **string** (*bytes*) – A string of bytes to be converted to a number.
- **type** (*int*) – See above table for valid options.

**Returns** Returns the integer or float value representing the byte string passed in.

**Return type** *int or float*

**s13.bytes\_to\_str** (*b*)

MicroPython does not support the latin1 character set, but this is equivalent to `str(s, 'latin1')` which converts a string to bytes without regard to unicode encoding

**s13.clear\_stats** ()

Clears Satlink’s counters, including transmission counts, reset counts, and more

**s13.command\_line** (*command, buf\_size=512*)

Issue commands to Satlink’s command line.

**Parameters**

- **command** (*str*) – command line function to execute, e.g. “Hello” “M1 LAST”

- **buf\_size** (*int*) – size of buffer for reply

**Returns** command line reply

**Return type** *str*

s13.**crc** (*data*, *polynomial*, *initial\_value=0*, *reflect=False*, *invert=False*, *reverse=False*)

Computes the CRC-16 over *data*. The following web site contains a lot of details on different CRC implementations, those details compromise the variables passed in to the `crc` function:

<http://protocoltool.sourceforge.net/CRC%20list.html>

Example:

```
>>> from s13 import *
>>> hex(crc("123456789", 0x1021))
'0x31c3'
```

### Parameters

- **data** (*str*) – A *str* or bytes to compute a CRC across.
- **polynomial** (*int*) – A 16-bit number that varies based on the type of CRC and determines how the CRC operates.
- **initial\_value** (*int*) – The initial value for the generator, typically 0, or 0xffff.
- **reflect** (*bool*) – There are two common versions of the CRC-16 algorithm, one that works by shifting the polynomial value left on each operation (normal), and one that works by shifting the reflected polynomial value right on each operation.
- **invert** (*bool*) – True to invert the result. Some variations require this.
- **reverse** (*bool*) – True to reverse the final byte order. Some variations require this.

**Returns** The computed 16-bit CRC value

**Return type** *int*

s13.**crc\_dnp** (*data*)

Computes the DNP CRC over *data*

**Parameters** **data** (*str*) – A *str* or bytes to compute a CRC across.

**Returns** The computed 16-bit CRC value

**Return type** *int*

s13.**crc\_kermit** (*data*)

Computes the Kermit CRC-CCITT over *data*

**Parameters** **data** (*str*) – A *str* or bytes to compute a CRC across.

**Returns** The computed 16-bit CRC value

**Return type** *int*

s13.**crc\_modbus** (*data*)

Computes the ModBus CRC-16 over *data*

**Parameters** **data** (*str*) – A *str* or bytes to compute a CRC across.

**Returns** The computed 16-bit CRC value

**Return type** *int*

`s13.crc_ssp(data)`

Computes the SSP CRC-16 over `data`

**Parameters** `data` (`str`) – A str or bytes to compute a CRC across.

**Returns** The computed 16-bit CRC value

**Return type** `int`

`s13.crc_xmodem(data)`

Computes the Xmodem CRC-CCITT over `data`

**Parameters** `data` (`str`) – A str or bytes to compute a CRC across.

**Returns** The computed 16-bit CRC value

**Return type** `int`

`s13.get_api_version()`

The version of the API.

**Returns** API version

**Return type** `float`

`s13.index()`

Retruns the index of the calling measurement or transmission or script task. For example, if a script hooked into measurement M7 were to call this function, it would return 7. If not called from a meas/tx/task, routine returns 1.

**Returns** index of caller

**Return type** `int`

`s13.internal_temp()`

Measures Satlink's internal temperature sensor :return: temperature in Celsius :rtype: float

`s13.is_being_tested()`

Tells you whether the function is being called as a part of a test operation.

**Returns** If a Python function is being tested by the customer, the value is True. If the function was invoked by Satlink as a part of normal operation, the value is False.

**Return type** `bool`

`s13.is_scheduled()`

Tells you whether the calling measurement/transmission/script task was scheduled.

**Returns** True if scheduled.s

**Return type** `bool`

`s13.lock()`

`lock()` and `unlock()` may be used to protect global variables from being corrupted by code designed to be run by multiple tasks at the same time. They do this by allowing only one task to proceed past the `lock()` statement at a time. Consider [Example 1](#) where a global variable is decremented. Without the lock, two tasks could interfere with each other and prevent the global variable from being decremented correctly. Similarly a section of code can be protected, such that only one task can enter that code at a time. If you need to protect a section of code that's more substantial than some quick operations, consider using `_thread.allocate_lock()` to create a unique lock variable instead, and reserve `lock()/unlock()` for the quick sections.

Example 1 - Simple lock/unlock:

```

from sl3 import lock, unlock
global my_global
lock()
my_global = my_global - 1
unlock()

```

Example 2 - Exception safe lock/unlock:

```

from sl3 import lock, unlock
# use try-finally to make sure the global lock is always released
# even if an exception occurs:
try:
    lock()
    call_my_function()
finally:
    unlock()

```

Example 3 - Exception safe using GLOBAL\_LOCK:

```

# try-finally is a little cumbersome, here's a simpler way to protect a section
↳of code
# in an exception-safe way:
from sl3 import GLOBAL_LOCK
with GLOBAL_LOCK:
    call_my_function()

```

Example 4 - Custom lock:

```

# in cases where long sections of code must be protected, it's possible to
↳introduce
# custom locks to avoid holding up code which use lock()/unlock():
import _thread
MY_LOCK = _thread.allocate_lock()
#
@MEASUREMENT
def shared_measurement(num):
    with MY_LOCK:
        return my_function(num)

```

`sl3.measure (meas_index, meas_action=2)`

Used to access sensor results via Measurements. Requires that a measurement be setup.

#### Parameters

- **meas\_index** – int, measurement index, 1 based, e.g. 4 for M4
- **meas\_action** –

#### One of the available constants indicating what action to take. Options:

**const** `READING_LAST` The reading produced when measurement was last made. Does not start a new measurement.

**const** `READING_LAST_SCHED` Like last, but returns the last scheduled (zs opposed to live/forced reading).

**const** `READING_MEAS_SYNC` Picks up the last reading synchronized to the schedule. Avoids extraneous readings. If the caller is scheduled for 12:15, this will use the measurement reading made at 12:15, waiting if need be. If system is not running, it will make a new measurement

**const** `READING_MEAS_FORCE` Initiates a new measurement every time

**Returns** The result of the sensor measurements

**Return type** *Reading*

`s13.output_control` (*port*, *turn\_on*)

Drives the digital output lines

**Parameters**

- **port** – which output to drive, string. options are ‘OUTPUT1’, ‘OUTPUT2’
- **turn\_on** – whether to turn on or off the output, Boolean

`s13.output_sample` (*port*)

Samples one of the output lines.

**Parameters** **port** – which output to drive, string. options are ‘OUTPUT1’, ‘OUTPUT2’

**Returns** Boolean indicating whether the output is on

`s13.power_control` (*port*, *turn\_on*)

Controls the various power output lines.

**Parameters**

- **port** – which port to drive, string: ‘PORT\_SW1’, ‘PORT\_SW2’, ‘PORT\_PROT12’, ‘RS232’, ‘VREF’, ‘PREF’
- **turn\_on** – Boolean indicating whether to turn on (True) or turn off (False) the power output

`s13.power_sample` (*port*)

Samples one of the various power output lines.

**Parameters** **port** – which port to sample, string: ‘PORT\_SW1’, ‘PORT\_SW2’, ‘PORT\_PROT12’, ‘RS232’, ‘VREF’, ‘PREF’

**Returns** Boolean indicating whether the power output is on

`s13.reset_count` ()

Tells you how many times Satlink has reset

**Returns** number of resets

**Return type** integer

`s13.seconds_since_midnight` ()

returns the number of seconds that have elapsed since midnight as an integer

`s13.setup_read` (*parameter*)

Read SL3 setup for specified parameter. Examples:

```
>>> From s13 import *
>>> setup_read("station name")
"Sutron Satlink 3"
>>> setup_read("recording")
"On"
```

**Parameters** **parameter** (*str*) – SL3 setup value to read

**Returns** parameter value in SL3

**Return type** *str*

`s13.setup_write` (*parameter*, *value*)

Change setup for specified parameter. Nothing is returned if write is successful. A `SetupError` will be raised if write failed.

Examples:

```
>>> From s13 import *
>>> setup_write("station name", "Little Creek")
>>> setup_write("M1 slope", 1.5)
```

#### Parameters

- **parameter** (*str*) – SL3 setup parameter to change.
- **value** – Value to change the setup to. Can be *str*, *bytes*, *float*, *int*.

**Returns** Nothing is returned if successful. Raises `SetupError` if the write failed or the setup did not take on the exact value.

`s13.sgn` (*number*)

Returns an integer indicating the sign of provided number

**Parameters** **number** (*integer*) – number whose sign we are interested in

**Returns** the sign of the number: 1 when above 0, returns 0 when 0, and returns -1 when negative

**Return type** *integer*

`s13.s13_time_to_seconds` (*str\_dt*)

Convert a string to time.

**Parameters** **str\_dt** (*str*) – SL3 date and time string YYYY/MM/DD HH:MM:SS or HH:MM:SS MM/DD/YYYY

**Returns** time in seconds since 1970, just like `utime.time()`

**Return type** *int*

`s13.str_to_bytes` (*s*)

MicroPython does not support the `latin1` character set, but this is equivalent to `bytes(s, 'latin1')` which converts bytes to string keeping the bytes exactly as-is without regard to unicode encoding

`s13.time_scheduled` ()

Returns the time the calling measurement or transmission or script task was scheduled to run.

**Returns** time in seconds since 1970, just like `utime.time()`

`s13.tls` ()

The `tls()` function returns a structure that may be used to store thread-local storage variables

In Python, when you create a global variable, every task, every custom format routine, and every measurement may access that variable. That can be useful when data must be shared, but sometimes separate storage is needed that is unique to the script task, formatter, or measurement That is what the `tls()` function provides.

**Creating a value in thread local storage is done by assigning a name in `tls` a value:** `tls().daily_rain = 100.0`

**Referencing a variable stored in thread local storage is just as simple:** `tls().daily_rain = tls().daily_rain + hourly_rain`

But how can we tell if a variable already exists so we can tell whether we need to initialize it? We can use `hasattr()`. The code below initializes daily rainfall:

```
if not hasattr(tls(), "daily_rain"):  
    tls().daily_rain = 0.0
```

We can also use `getattr()` to retrieve a value regardless of whether one exists in `tls()`. The code below will retrieve a value for `slope` if one has been stored in `tls()`, otherwise use `1.0`:

```
slope = getattr(tls(), "slope", 1.0)
```

Finally, `tls` is not limited to simple values. Any data-type that can be assigned to a variable may be stored including strings, lists, and other objects.

**s13.trigger\_meas** (*meas\_index*)

Initiates a live reading of said measurement. Does not wait for measurement to complete.

**Parameters** `meas_index` – measurement index int, 1 to 32. Use 0 to trigger all measurement.

**Returns** None

**s13.trigger\_script\_task** (*task\_index*)

Initiates said script task. Does not wait for completion.

**Parameters** `task_index` – Script task index int, 1 to 8. Use 0 to trigger all tasks.

**Returns** None

**s13.trigger\_tx** (*tx\_index*)

Initiates a transmission. Does not wait for transmission to complete.

**Parameters** `tx_index` – Index of transmission. You may not trigger all transmissions by passing in 0.

**Returns** None

**s13.unlock** ()

`lock()` and `unlock()` are typically used together to protect global variables.

**s13.ver** ()

Tells the Satlink software version:

```
>>> ver()  
[8, 10, 2578]
```

**Returns** major, minor, and revision

**Return type** *tuple*

**s13.ver\_boot** ()

Tells the Satlink bootloader software version:

```
>>> ver_boot()  
[8, 10, 0]
```

**Returns** major, minor, and revision. revision is always 0 for bootloader

**Return type** *tuple*

**s13.wait\_for** (*device, pattern*)

Wait for a pattern of bytes to be received over a communications device

**Parameters**



- **device** (`Serial`) – Any Serial, File, or Stream object with a `read()` method.
- **pattern** (`str`) – A string to look for in the input stream of the device.

**Returns** True if a match was found or False if a timeout occurred.

**Return type** *bool*

**patterns::**

- a “?” in the pattern will match any single character
- a “\*” will match anything (ie “A\*G” would match “ABCDEFGG”)
- control characters may be embedded with “^” (ie “^C” would match a ctrl-c)
- special characters may also be embedded with “” (ie “ÿ” would match a 255 ascii)
- any of the special codes may be matched by escaping with a “” (ex: “\*”) `wait_for` is written to accept either byte or string oriented devices and patterns



## THE MICROPYTHON LANGUAGE

MicroPython aims to implement the Python 3.4 standard, and most of the features of MicroPython are identical to those described by the documentation at [docs.python.org](https://docs.python.org).

Differences to standard Python as well as additional features of MicroPython are described in the sections here.

### 3.1 Overview of MicroPython Differences from Standard Python

MicroPython does not support the entire Python standard library. If a module is missing it will be due to the inapplicability of that module for use in an embedded controller. High memory consumption (e.g. `sqlite3`) or a lack of a certain required hardware feature (e.g. multiprocessing) are reasons that some modules can not be implemented for some microcontrollers. The full list of standard python libraries can be found here: [Python 3.4 standard lib](#)

There are several differences between CPython3 (considered to be a reference implementation of the Python3 language) and MicroPython. These differences are classified into 3 categories, each category having a different status regarding the possibility that items belonging to it will change.

#### 3.1.1 Differences By Design

MicroPython is intended for constrained environments, in particular, microcontrollers, which have orders of magnitude less performance and memory than “desktop” systems on which CPython3 runs. This means that MicroPython must be designed with this constrained environment in mind, leaving out features which simply won’t fit, or won’t scale, with target systems. “By design” differences are unlikely to change.

1. MicroPython does not ship with an extensive standard library of modules. It’s not possible and does not make sense, to provide the complete CPython3 library. Many modules are not usable or useful in the context of embedded systems, and there is not enough memory to deploy the entire library on small devices. So MicroPython takes the minimalist approach - only core datatypes (plus modules specific to particular hardware) are included with the interpreter, and all the rest is left as 3rd-party dependencies for particular user applications. The [micropython-lib](#) project provides a non-monolithic standard library for MicroPython ([forum](#))
2. Unlike CPython3, which uses reference-counting, MicroPython uses garbage collection as the primary means of memory management.
3. MicroPython does not implement complete CPython object data model, but only a subset of it. Advanced usages of multiple inheritance, `__new__` method may not work. Method resolution order is different (#525). Metaclasses are not supported (at least yet).
4. MicroPython design is not based around descriptor objects. So far, we were able to implement all native Python features (like properties) without explicit descriptors. Descriptors are considered “overdynamic” feature, and conflicts with the aim of being fast and efficient. However, simplified support for descriptors is now implemented, as an opt-in feature.

5. By virtue of being “micro”, MicroPython implements only subset of functionality and parameters of particular functions and classes. Each specific issue may be treated as “implementation difference” and resolved, but there unlikely will ever be 100% coverage of CPython features.
6. By virtue of being “micro”, MicroPython supports only minimal subset of introspection and reflection features (such as object names, docstrings, etc.). Each specific feature may be treated as “implementation difference” and resolved, but there unlikely will ever be 100% coverage of CPython features.
7. `print()` function does not check for recursive data structures in the same way CPython does. There’s however checks for stack usage, so printing recursive data structure won’t lead to crash due to stack overflow. Note that it’s possible to implement more CPython-like handling of recursive data structures on the Python application level. Do this by writing a function which keeps the history of each object visited, and override the builtin `print()` with your custom function. Such an implementation may of course use a lot of memory, which is why it’s not implemented at the MicroPython level.
8. MicroPython optimizes local variable handling and does not record or provide any introspection info for them, e.g. `locals()` doesn’t have entries for locals.

### 3.1.2 Implementation Differences

Some features don’t cater for constrained systems, and at the same time are not easy to implement efficiently, or at all. Such features are known as “implementation differences” and some of them are potentially the subject for future development (after corresponding discussion and consideration). Note that many of them would affect the size and performance of any given MicroPython implementation, so sometimes not implementing a specific feature is identified by MicroPython’s target usage.

1. Unicode support is work in progress. It is based on internal representation using UTF-8. Strings containing Unicode escapes in the `xNN`, `uNNNN`, and `U000NNNNN` forms are fully supported; `N{...}` is not supported. #695.
2. Object finalization (`__del__()` method) is supported for builtin types, but not yet user classes. This is tracked by #245.
3. Subclassing of builtin types is partially implemented, but there may be various differences and compatibility issues with CPython. #401
4. Buffered I/O streams (`io.TextIOWrapper` and superclasses) are not supported.
5. `async def` keyword is implemented with the following simplifications: there’s no separate “coroutine” object type, `async def` just defines a generator function without requirement for “yield” or “yield from” to appear in function body. Usage of “yield” or “yield from” in `async def` function is not detected or banned.
6. `async with` should be equivalent to [PEP492 description](#).
7. `async for` should be equivalent to [PEP492 description](#).
8. There’s no support for “Future-like objects” with `__await__` method. `await` can be used only on coroutines and generators (not “Future-like objects”). It’s also just a syntactic sugar for “yield from” and thus accepts iterables and iterators, which are not allowed in CPython.
9. Instance `__dict__` support is optional (disabled in many ports) and read-only, so `foo.__dict__[‘bar’] = 23` or `foo.__dict__.update(‘bar’: 23)` does not work. #1757 #2139

### 3.1.3 Known Issues

Known issues are essentially bugs, misfeatures, and omissions considered such, and scheduled to be fixed. So, ideally any entry here should be accompanied by bug ticket reference. But note that these known issues will have different priorities, especially within wider development process. So if you are actually affected by some issue, please add

details of your case to the ticket (or open it if does not yet exist) to help planning. Submitting patches is even more productive. (Please note that the list of not implemented modules/classes include only those which are considered very important to implement; per the above, MicroPython does not provide full standard library in general.)

1. Currently, it's not possible to override `sys.stdin`, `sys.stdout`, `sys.stderr` (for efficiency, they are stored in read-only memory).
2. `str.format()` may miss few advanced/obscure features (but otherwise almost completely implemented). [#407](#), [#574](#)
3. Instead of `re` module, minimized “`ure`” module is provided is subset of functionality. `micropython-lib` also offers more complete implementation based on PCRE engine, for “`unix`” port [#13](#)
4. Only beginning of `io` module and class hierarchy exists so far.
5. `collections.deque` class is not implemented.
6. Container slice assignment/deletion is only partially implemented. [#509](#)
7. 3-argument slicing is only partially implemented.
8. Only basic support for `__new__` method is available [#606](#), [#622](#).

## 3.2 Code examples of how MicroPython differs from Standard Python with work-arounds

The operations listed in this section produce conflicting results in MicroPython when compared to standard Python.

### 3.2.1 Syntax

Generated Tue 13 Jun 2017 17:30:45 UTC

#### Spaces

**uPy requires spaces between literal numbers and keywords, CPy doesn't**

Sample code:

```
try:
    print(eval('land 0'))
except SyntaxError:
    print('Should have worked')
try:
    print(eval('lor 0'))
except SyntaxError:
    print('Should have worked')
try:
    print(eval('lif lelse 0'))
except SyntaxError:
    print('Should have worked')
```

CPy output:	uPy output:
<pre>0 1 1</pre>	<pre>Should have worked Should have worked Should have worked</pre>

## Unicode

### Unicode name escapes are not implemented

Sample code:

```
print("\N{LATIN SMALL LETTER A}")
```

CPy output:	uPy output:
a	<code>NotImplementedError: unicode name escapes</code>

## 3.2.2 Core Language

Generated Tue 13 Jun 2017 17:30:45 UTC

### Error messages for methods may display unexpected argument counts

**Cause:** MicroPython counts “self” as an argument.

**Workaround:** Interpret error messages with the information above in mind.

Sample code:

```
try:
    [].append()
except Exception as e:
    print(e)
```

CPy output:	uPy output:
<pre>append() takes exactly one argument (0 ↳given)</pre>	<pre>function takes 2 positional arguments ↳but 1 were given</pre>

### Method Resolution Order (MRO) is not compliant with CPython

Sample code:

```
class Foo:
    def __str__(self):
        return "Foo"

class C(tuple, Foo):
    pass
```

```
t = C((1, 2, 3))
print(t)
```

CPy output:	uPy output:
Foo	(1, 2, 3)

## Classes

Special method `__del__` not implemented for user-defined classes (this is true for standard MicroPython but has been fixed in the Satlink 3, also deleting an object will finalize it immediately.)

**Workaround:** None needed for Satlink3

Sample code:

```
import gc

class Foo():
    def __del__(self):
        print('__del__')

f = Foo()
del f

gc.collect()
```

CPy output:	uPy output:
<code>__del__</code>	

**When inheriting from multiple classes `super()` only calls one class**

**Cause:** Depth first non-exhaustive method resolution order

Sample code:

```
class A:
    def __init__(self):
        print("A.__init__")

class B(A):
    def __init__(self):
        print("B.__init__")
        super().__init__()

class C(A):
    def __init__(self):
        print("C.__init__")
        super().__init__()

class D(B, C):
```

```
def __init__(self):
    print("D.__init__")
    super().__init__()

D()
```

CPy output:	uPy output:
D.__init__ B.__init__ C.__init__ A.__init__	D.__init__ B.__init__ A.__init__

**Calling super() getter property in subclass will return a property object, not the value**

Sample code:

```
class A:
    @property
    def p(self):
        return {"a":10}

class AA(A):
    @property
    def p(self):
        return super().p

a = AA()
print(a.p)
```

CPy output:	uPy output:
{'a': 10}	<property>

**Functions**

**Unpacking function arguments in non-last position isn't detected as an error**

**Workaround:** The syntax below is invalid, never use it in applications.

Sample code:

```
print(*(1, 2), 3)
```

CPy output:	uPy output:
1 2 3	3 1 2

**User-defined attributes for functions are not supported**

**Cause:** MicroPython is highly optimized for memory usage.



**Workaround:** Use external dictionary, e.g. `FUNC_X[f] = 0`.

Sample code:

```
def f():
    pass

f.x = 0
print(f.x)
```

CPy output:	uPy output:
0	Traceback (most recent call last): File "<stdin>", line 10, in <module> AttributeError: 'function' object has no attribute 'x'

## Generator

**Context manager `__exit__()` not called in a generator which does not run to completion**

Sample code:

```
class foo(object):
    def __enter__(self):
        print('Enter')
    def __exit__(self, *args):
        print('Exit')

def bar(x):
    with foo():
        while True:
            x += 1
            yield x

def func():
    g = bar(0)
    for _ in range(3):
        print(next(g))

func()
```

CPy output:	uPy output:
Enter 1 2 3 Exit	Enter 1 2 3

## import

**`__path__` attribute of a package has a different type (single string instead of list of strings) in MicroPython**

**Cause:** MicroPython doesn't support namespace packages split across filesystem. Beyond that, MicroPython's import system is highly optimized for minimal memory usage.

**Workaround:** Details of import handling is inherently implementation dependent. Don't rely on such details in portable applications.

Sample code:

```
import modules

print(modules.__path__)
```

CPy output:	uPy output:
<pre>[ ↪ 'C:\\Users\\jon\\AppData\\Local\\Temp\\sl3- ↪ build\\src\\MicroPython\\test\\packages\\cpydiff\\modules ↪ ']</pre>	<pre>../test/packages/cpydiff//modules</pre>

**Failed to load modules are still registered as loaded**

**Cause:** To make module handling more efficient, it's not wrapped with exception handling.

**Workaround:** Test modules before production use; during development, test using LinkComm, issue "script test" at the command prompt, or from the repl `del sys.modules["name"]` will work. Worst-case reset the board.

Sample code:

```
import sys

try:
    from modules import foo
except NameError as e:
    print(e)

try:
    from modules import foo
    print('Should not get here')
except NameError as e:
    print(e)
```

CPy output:	uPy output:
<pre>foo name 'xxx' is not defined foo name 'xxx' is not defined</pre>	<pre>foo name 'xxx' is not defined Should not get here</pre>

**MicroPython doesn't support namespace packages split across filesystem.**

**Cause:** MicroPython's import system is highly optimized for simplicity, minimal memory usage, and minimal filesystem search overhead.

**Workaround:** Don't install modules belonging to the same namespace package in different directories. For MicroPython, it's recommended to have at most 3-component module search paths: for your current application, per-user (writable), system-wide (non-writable).

Sample code:

```
import sys
sys.path.append(sys.path[1] + "/modules")
sys.path.append(sys.path[1] + "/modules2")

import subpkg.foo
import subpkg.bar

print("Two modules of a split namespace package imported")
```

CPy output:	uPy output:
Two modules of a split namespace package, ↳ imported	Traceback (most recent call last): File "<stdin>", line 12, in <module> ImportError: no module named 'subpkg.bar'

### 3.2.3 Builtin Types

Generated Tue 13 Jun 2017 17:30:45 UTC

#### Exception

##### Exception chaining not implemented

Sample code:

```
try:
    raise TypeError
except TypeError:
    raise ValueError
```

CPy output:	uPy output:
Traceback (most recent call last): File "<stdin>", line 8, in <module> TypeError  During handling of the above exception, ↳ another exception occurred:  Traceback (most recent call last): File "<stdin>", line 10, in <module> ValueError	Traceback (most recent call last): File "<stdin>", line 10, in <module> ValueError:

##### Assign instance variable to exception

Sample code:

```
e = Exception()
e.x = 0
print(e.x)
```

CPy output:	uPy output:
0	Traceback (most recent call last): File "<stdin>", line 8, in <module> AttributeError: 'Exception' object has ↳no attribute 'x'

**While loop guards will obscure exception line number reporting due to being optimised onto the end of the code block**

Sample code:

```
l = ["-foo", "-bar"]

i = 0
while l[i][0] == "-":
    print("iter")
    i += 1
```

CPy output:	uPy output:
iter iter Traceback (most recent call last): File "<stdin>", line 10, in <module> IndexError: list index out of range	iter iter Traceback (most recent call last): File "<stdin>", line 12, in <module> IndexError: list index out of range

**Exception.\_\_init\_\_ raises TypeError if overridden and called by subclass**

Sample code:

```
class A(Exception):
    def __init__(self):
        Exception.__init__(self)

a = A()
```

CPy output:	uPy output:
	Traceback (most recent call last): File "<stdin>", line 11, in <module> File "<stdin>", line 9, in __init__ TypeError: argument should be a ↳'Exception' not a 'A'

**bytearray**

**Array slice assignment with unsupported RHS**

Sample code:

```
b = bytearray(4)
b[0:1] = [1, 2]
print(b)
```

CPy output:	uPy output:
<code>bytearray(b'\x01\x02\x00\x00')</code>	Traceback (most recent call last): File "<stdin>", line 8, in <module> NotImplementedError: array/bytes_ ↪required on right side

## bytes

### bytes(...) with keywords not implemented

**Workaround:** Input the encoding format directly. eg. `print(bytes('abc', 'utf-8'))`

Sample code:

```
print(bytes('abc', encoding='utf8'))
```

CPy output:	uPy output:
<code>b'abc'</code>	Traceback (most recent call last): File "<stdin>", line 7, in <module> NotImplementedError: keyword argument(s)_ ↪ <b>not</b> yet implemented - use normal args_ ↪instead

### Bytes subscr with step != 1 not implemented

Sample code:

```
print(b'123'[0:3:2])
```

CPy output:	uPy output:
<code>b'13'</code>	Traceback (most recent call last): File "<stdin>", line 7, in <module> NotImplementedError: only slices <b>with</b> _ ↪step=1 (aka <b>None</b> ) are supported

## float

### uPy and CPython outputs formats differ

Sample code:

```
print('%1g' % -9.9)
print('%1e' % 9.99)
print('%1e' % 0.999)
```

CPy output:	uPy output:
<pre>-1e+01 1.0e+01 1.0e+00</pre>	<pre>-10 1.00e+01 1.00e-00</pre>

## int

### No int conversion for int-derived types available

Sample code:

```
class A(int):
    __add__ = lambda self, other: A(int(self) + other)

a = A(42)
print(a+a)
```

CPy output:	uPy output:
<pre>84</pre>	<pre>Traceback (most recent call last):   File "&lt;stdin&gt;", line 11, in &lt;module&gt;   File "&lt;stdin&gt;", line 8, in &lt;lambda&gt; TypeError: can't convert A to int</pre>

### Incorrect error message when passing float into to\_bytes

Sample code:

```
try:
    int('1').to_bytes(1.0)
except TypeError as e:
    print(e)
```

CPy output:	uPy output:
<pre>integer argument expected, got float</pre>	<pre>function missing 1 required positional_ ↳arguments</pre>

## list

### List delete with step != 1 not implemented

Sample code:

```
l = [1, 2, 3, 4]
del l[0:4:2]
print(l)
```

CPy output:	uPy output:
<pre>[2, 4]</pre>	<pre>Traceback (most recent call last):   File "&lt;stdin&gt;", line 8, in &lt;module&gt; NotImplementedError:</pre>

**List slice-store with non-iterable on RHS is not implemented**

**Cause:** RHS is restricted to be a tuple or list

**Workaround:** Use `list(<iter>)` on RHS to convert the iterable to a list

Sample code:

```
l = [10, 20]
l[0:1] = range(4)
print(l)
```

CPy output:	uPy output:
<pre>[0, 1, 2, 3, 20]</pre>	<pre>Traceback (most recent call last):   File "&lt;stdin&gt;", line 8, in &lt;module&gt; TypeError: object 'range' is not a tuple, ↳or list</pre>

**List store with step != 1 not implemented**

Sample code:

```
l = [1, 2, 3, 4]
l[0:4:2] = [5, 6]
print(l)
```

CPy output:	uPy output:
<pre>[5, 2, 6, 4]</pre>	<pre>Traceback (most recent call last):   File "&lt;stdin&gt;", line 8, in &lt;module&gt; NotImplementedError:</pre>

**str****UnicodeDecodeError not raised when expected**

Sample code:

```
try:
    print(repr(str(b"\xa1\x80", 'utf8')))
    print('Should not get here')
except UnicodeDecodeError:
    print('UnicodeDecodeError')
```

CPy output:	uPy output:
UnicodeDecodeError	'\u0840' Should <b>not</b> get here

**Start/end indices such as str.endswith(s, start) not implemented**

Sample code:

```
print('abc'.endswith('c', 1))
```

CPy output:	uPy output:
<b>True</b>	Traceback (most recent call last): File "<stdin>", line 7, in <module> NotImplementedError: start/end indices

**Attributes/subscr not implemented**

Sample code:

```
print('{a[0]}'.format(a=[1, 2]))
```

CPy output:	uPy output:
1	Traceback (most recent call last): File "<stdin>", line 7, in <module> NotImplementedError: attributes <b>not</b> supported yet

**str(...) with keywords not implemented**

**Workaround:** Input the encoding format directly. eg print(bytes('abc', 'utf-8'))

Sample code:

```
print(str(b'abc', encoding='utf8'))
```

CPy output:	uPy output:
abc	Traceback (most recent call last): File "<stdin>", line 7, in <module> NotImplementedError: keyword argument(s) <b>not</b> yet implemented - use normal args instead

**None as first argument for rsplit such as str.rsplit(None, n) not implemented**

Sample code:



```
print('a a a'.rsplit(None, 1))
```

CPy output:	uPy output:
['a a', 'a']	Traceback (most recent call last): File "<stdin>", line 7, in <module> NotImplementedError: rsplit( <b>None</b> ,n)

### Instance of a subclass of str cannot be compared for equality with an instance of a str

Sample code:

```
class S(str):
    pass

s = S('hello')
print(s == 'hello')
```

CPy output:	uPy output:
<b>True</b>	<b>False</b>

### Subscript with step != 1 is not yet implemented

Sample code:

```
print('abcdefghi'[0:9:2])
```

CPy output:	uPy output:
acegi	Traceback (most recent call last): File "<stdin>", line 7, in <module> NotImplementedError: only slices with ↳step=1 (aka <b>None</b> ) are supported

## tuple

### Tuple load with step != 1 not implemented

Sample code:

```
print((1, 2, 3, 4)[0:4:2])
```

CPy output:	uPy output:
(1, 3)	Traceback (most recent call last): File "<stdin>", line 7, in <module> NotImplementedError: only slices with ↳step=1 (aka <b>None</b> ) are supported

### 3.2.4 Modules

Generated Tue 13 Jun 2017 17:30:45 UTC

#### array

##### Looking for integer not implemented

Sample code:

```
import array
print(1 in array.array('B', b'12'))
```

CPy output:	uPy output:
<code>False</code>	Traceback (most recent call last): File "<stdin>", line 8, in <module> NotImplementedError:

##### Array deletion not implemented

Sample code:

```
import array
a = array.array('b', (1, 2, 3))
del a[1]
print(a)
```

CPy output:	uPy output:
<code>array('b', [1, 3])</code>	Traceback (most recent call last): File "<stdin>", line 9, in <module> TypeError: 'array' object does not ↳support item deletion

##### Subscript with step != 1 is not yet implemented

Sample code:

```
import array
a = array.array('b', (1, 2, 3))
print(a[3:2:2])
```

CPy output:	uPy output:
<code>array('b')</code>	Traceback (most recent call last): File "<stdin>", line 9, in <module> NotImplementedError: only slices with ↳step=1 (aka <b>None</b> ) are supported

## deque

### Deque not implemented

**Workaround:** Use regular queues or lists creatively

Sample code:

```
import collections
D = collections.deque()
print(D)
```

CPy output:	uPy output:
deque([])	Traceback (most recent call last): File "<stdin>", line 8, in <module> AttributeError: 'module' object has no attribute 'deque'

## json

### JSON module does not throw exception when object is not serialisable

Sample code:

```
import json
a = bytes(x for x in range(256))
try:
    z = json.dumps(a)
    x = json.loads(z)
    print('Should not get here')
except TypeError:
    print('TypeError')
```

CPy output:	uPy output:
TypeError	Should <b>not</b> get here

## struct

### Struct pack with too few args, not checked by uPy

Sample code:

```
import struct
try:
    print(struct.pack('bb', 1))
    print('Should not get here')
except:
    print('struct.error')
```

CPy output:	uPy output:
<code>struct.error</code>	<code>b'\x01\x00'</code> Should <b>not</b> get here

### Struct pack with too many args, not checked by uPy

Sample code:

```
import struct
try:
    print(struct.pack('bb', 1, 2, 3))
    print('Should not get here')
except:
    print('struct.error')
```

CPy output:	uPy output:
<code>struct.error</code>	<code>b'\x01\x02'</code> Should <b>not</b> get here

## sys

Override `sys.stdin`, `sys.stdout` and `sys.stderr`. Impossible as they are stored in read-only memory.

Sample code:

```
import sys
sys.stdin = None
print(sys.stdin)
```

CPy output:	uPy output:
<code>None</code>	Traceback (most recent call last): File "<stdin>", line 8, in <module> AttributeError: 'module' object has no <u>attribute 'stdin'</u>

## 3.3 The MicroPython Interactive Interpreter Mode (aka REPL)

This section covers some characteristics of the MicroPython Interactive Interpreter Mode. A commonly used term for this is REPL (read-eval-print-loop) which will be used to refer to this interactive prompt.

### 3.3.1 Auto-indent

When typing python statements which end in a colon (for example `if`, `for`, `while`) then the prompt will change to three dots (...) and the cursor will be indented by 4 spaces. When you press return, the next line will continue at the same

level of indentation for regular statements or an additional level of indentation where appropriate. If you press the backspace key then it will undo one level of indentation.

If your cursor is all the way back at the beginning, pressing RETURN will then execute the code that you've entered. The following shows what you'd see after entering a for statement (the underscore shows where the cursor winds up):

```
>>> for i in range(3):
...     _
```

If you then enter an if statement, an additional level of indentation will be provided:

```
>>> for i in range(30):
...     if i > 3:
...         _
```

Now enter `break` followed by RETURN and press BACKSPACE:

```
>>> for i in range(30):
...     if i > 3:
...         break
...     _
```

Finally type `print(i)`, press RETURN, press BACKSPACE and press RETURN again:

```
>>> for i in range(30):
...     if i > 3:
...         break
...     print(i)
...
0
1
2
3
>>>
```

Auto-indent won't be applied if the previous two lines were all spaces. This means that you can finish entering a compound statement by pressing RETURN twice, and then a third press will finish and execute.

### 3.3.2 Auto-completion

While typing a command at the REPL, if the line typed so far corresponds to the beginning of the name of something, then pressing TAB will show possible things that could be entered. For example type `m` and press TAB and it should expand to `machine`. Enter a dot `.` and press TAB again. You should see something like:

```
>>> machine.
__name__          info                unique_id          reset
bootloader       freq                rng                idle
sleep            deepsleep          disable_irq        enable_irq
Pin
```

The word will be expanded as much as possible until multiple possibilities exist. For example, type `machine.Pin`. AF3 and press TAB and it will expand to `machine.Pin.AF3_TIM`. Pressing TAB a second time will show the possible expansions:

```
>>> machine.Pin.AF3_TIM
AF3_TIM10        AF3_TIM11          AF3_TIM8           AF3_TIM9
>>> machine.Pin.AF3_TIM
```

### 3.3.3 Interrupting a running program

You can interrupt a running program by pressing Ctrl-C. This will raise a KeyboardInterrupt which will bring you back to the REPL, providing your program doesn't intercept the KeyboardInterrupt exception.

For example:

```
>>> for i in range(1000000):
...     print(i)
...
0
1
2
3
...
6466
6467
6468
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt:
>>>
```

### 3.3.4 Paste Mode

If you want to paste some code into your terminal window, the auto-indent feature will mess things up. For example, if you had the following python code:

```
def foo():
    print('This is a test to show paste mode')
    print('Here is a second line')
foo()
```

and you try to paste this into the normal REPL, then you will see something like this:

```
>>> def foo():
...     print('This is a test to show paste mode')
...         print('Here is a second line')
...     foo()
...
  File "<stdin>", line 3
IndentationError: unexpected indent
```

If you press Ctrl-E, then you will enter paste mode, which essentially turns off the auto-indent feature, and changes the prompt from >>> to ===". For example:

```
>>>
paste mode; Ctrl-C to cancel, Ctrl-D to finish
=== def foo():
===     print('This is a test to show paste mode')
===     print('Here is a second line')
=== foo()
===
This is a test to show paste mode
Here is a second line
>>>
```

Paste Mode allows blank lines to be pasted. The pasted text is compiled as if it were a file. Pressing Ctrl-D exits paste mode and initiates the compilation.

### 3.3.5 Soft Reset

A soft reset will reset the python interpreter, but tries not to reset the method by which you're connected to the MicroPython board (USB-serial, or Wifi).

You can perform a soft reset from the REPL by pressing Ctrl-D, or from your python code by executing:

```
raise SystemExit
```

For example, if you reset your MicroPython board, and you execute a `dir()` command, you'd see something like this:

```
>>> dir()
['__name__', 'pyb']
```

Now create some variables and repeat the `dir()` command:

```
>>> i = 1
>>> j = 23
>>> x = 'abc'
>>> dir()
['j', 'x', '__name__', 'pyb', 'i']
>>>
```

Now if you enter Ctrl-D, and repeat the `dir()` command, you'll see that your variables no longer exist:

```
PYB: sync filesystems
PYB: soft reboot
MicroPython v1.5-51-g6f70283-dirty on 2015-10-30; PYBv1.0 with STM32F405RG
Type "help()" for more information.
>>> dir()
['__name__', 'pyb']
>>>
```

### 3.3.6 The special variable `_` (underscore)

When you use the REPL, you may perform computations and see the results. MicroPython stores the results of the previous statement in the variable `_` (underscore). So you can use the underscore to save the result in a variable. For example:

```
>>> 1 + 2 + 3 + 4 + 5
15
>>> x = _
>>> x
15
>>>
```

### 3.3.7 Raw Mode

Raw mode is not something that a person would normally use. It is intended for programmatic use. It essentially behaves like paste mode with echo turned off.

Raw mode is entered using Ctrl-A. You then send your python code, followed by a Ctrl-D. The Ctrl-D will be acknowledged by 'OK' and then the python code will be compiled and executed. Any output (or errors) will be sent back. Entering Ctrl-B will leave raw mode and return the the regular (aka friendly) REPL.

The `tools/pyboard.py` program uses the raw REPL to execute python files on the MicroPython board.

### 3.4 Maximising Python Speed

This tutorial describes ways of improving the performance of MicroPython code. Optimisations involving other languages are covered elsewhere, namely the use of modules written in C and the MicroPython inline ARM Thumb-2 assembler.

The process of developing high performance code comprises the following stages which should be performed in the order listed.

- Design for speed.
- Code and debug.

Optimisation steps:

- Identify the slowest section of code.
- Improve the efficiency of the Python code.
- Use the native code emitter.
- Use the viper code emitter.

#### 3.4.1 Designing for speed

Performance issues should be considered at the outset. This involves taking a view on the sections of code which are most performance critical and devoting particular attention to their design. The process of optimisation begins when the code has been tested: if the design is correct at the outset optimisation will be straightforward and may actually be unnecessary.

#### Algorithms

The most important aspect of designing any routine for performance is ensuring that the best algorithm is employed. This is a topic for textbooks rather than for a MicroPython guide but spectacular performance gains can sometimes be achieved by adopting algorithms known for their efficiency.

#### RAM Allocation

To design efficient MicroPython code it is necessary to have an understanding of the way the interpreter allocates RAM. When an object is created or grows in size (for example where an item is appended to a list) the necessary RAM is allocated from a block known as the heap. This takes a significant amount of time; further it will on occasion trigger a process known as garbage collection which can take several milliseconds.

Consequently the performance of a function or method can be improved if an object is created once only and not permitted to grow in size. This implies that the object persists for the duration of its use: typically it will be instantiated in a class constructor and used in various methods.

This is covered in further detail *Controlling garbage collection* below.



## Buffers

An example of the above is the common case where a buffer is required, such as one used for communication with a device. A typical driver will create the buffer in the constructor and use it in its I/O methods which will be called repeatedly.

The MicroPython libraries typically provide support for pre-allocated buffers. For example, objects which support stream interface (e.g., file or UART) provide `read()` method which allocate new buffer for read data, but also a `readinto()` method to read data into an existing buffer.

## Floating Point

Some MicroPython ports allocate floating point numbers on heap. Some other ports may lack dedicated floating-point coprocessor, and perform arithmetic operations on them in “software” at considerably lower speed than on integers. Where performance is important, use integer operations and restrict the use of floating point to sections of the code where performance is not paramount. For example, capture ADC readings as integers values to an array in one quick go, and only then convert them to floating-point numbers for signal processing.

## Arrays

Consider the use of the various types of array classes as an alternative to lists. The `array` module supports various element types with 8-bit elements supported by Python’s built in `bytes` and `bytearray` classes. These data structures all store elements in contiguous memory locations. Once again to avoid memory allocation in critical code these should be pre-allocated and passed as arguments or as bound objects.

When passing slices of objects such as `bytearray` instances, Python creates a copy which involves allocation of the size proportional to the size of slice. This can be alleviated using a `memoryview` object. `memoryview` itself is allocated on heap, but is a small, fixed-size object, regardless of the size of slice it points too.

```
ba = bytearray(10000) # big array
func(ba[30:2000])    # a copy is passed, ~2K new allocation
mv = memoryview(ba) # small object is allocated
func(mv[30:2000])   # a pointer to memory is passed
```

A `memoryview` can only be applied to objects supporting the buffer protocol - this includes arrays but not lists. Small caveat is that while `memoryview` object is live, it also keeps alive the original buffer object. So, a `memoryview` isn’t a universal panacea. For instance, in the example above, if you are done with 10K buffer and just need those bytes 30:2000 from it, it may be better to make a slice, and let the 10K buffer go (be ready for garbage collection), instead of making a long-living `memoryview` and keeping 10K blocked for GC.

Nonetheless, `memoryview` is indispensable for advanced preallocated buffer management. `.readinto()` method discussed above puts data at the beginning of buffer and fills in entire buffer. What if you need to put data in the middle of existing buffer? Just create a `memoryview` into the needed section of buffer and pass it to `.readinto()`.

### 3.4.2 Identifying the slowest section of code

This is a process known as profiling and is covered in textbooks and (for standard Python) supported by various software tools. For the type of smaller embedded application likely to be running on MicroPython platforms the slowest function or method can usually be established by judicious use of the timing `ticks` group of functions documented [here](#). Code execution time can be measured in ms, us, or CPU cycles.

The following enables any function or method to be timed by adding an `@timed_function` decorator:

```
def timed_function(f, *args, **kwargs):
    myname = str(f).split(' ')[1]
    def new_func(*args, **kwargs):
        t = time.ticks_us()
        result = f(*args, **kwargs)
        delta = time.ticks_diff(t, time.ticks_us())
        print('Function {} Time = {:.3f}ms'.format(myname, delta/1000))
        return result
    return new_func
```

### 3.4.3 MicroPython code improvements

#### The const() declaration

MicroPython provides a `const()` declaration. This works in a similar way to `#define` in C in that when the code is compiled to bytecode the compiler substitutes the numeric value for the identifier. This avoids a dictionary lookup at runtime. The argument to `const()` may be anything which, at compile time, evaluates to an integer e.g. `0x100` or `1 << 8`.

#### Caching object references

Where a function or method repeatedly accesses objects performance is improved by caching the object in a local variable:

```
class foo(object):
    def __init__(self):
        ba = bytearray(100)
    def bar(self, obj_display):
        ba_ref = self.ba
        fb = obj_display.framebuffer
        # iterative code using these two objects
```

This avoids the need repeatedly to look up `self.ba` and `obj_display.framebuffer` in the body of the method `bar()`.

#### Controlling garbage collection

When memory allocation is required, MicroPython attempts to locate an adequately sized block on the heap. This may fail, usually because the heap is cluttered with objects which are no longer referenced by code. If a failure occurs, the process known as garbage collection reclaims the memory used by these redundant objects and the allocation is then tried again - a process which can take several milliseconds.

There are benefits in pre-empting this by periodically issuing `gc.collect()`. Firstly doing a collection before it is actually required is quicker - typically on the order of 1ms if done frequently. Secondly you can determine the point in code where this time is used rather than have a longer delay occur at random points, possibly in a speed critical section. Finally performing collections regularly can reduce fragmentation in the heap. Severe fragmentation can lead to non-recoverable allocation failures.

#### Accessing hardware directly

This comes into the category of more advanced programming and involves some knowledge of the target MCU. Consider the example of toggling an output pin on the Pyboard. The standard approach would be to write

```
mypin.value(mypin.value() ^ 1) # mypin was instantiated as an output pin
```

This involves the overhead of two calls to the `Pin` instance's `value()` method. This overhead can be eliminated by performing a read/write to the relevant bit of the chip's GPIO port output data register (`odr`). To facilitate this the `stm` module provides a set of constants providing the addresses of the relevant registers. A fast toggle of pin `P4` (CPU pin `A14`) - corresponding to the green LED - can be performed as follows:

```
BIT14 = const(1 << 14)
stm.mem16[stm.GPIOA + stm.GPIO_ODR] ^= BIT14
```

### 3.4.4 The Native code emitter

This causes the MicroPython compiler to emit ARM native opcodes rather than bytecode. It covers the bulk of the Python language so most functions will require no adaptation (but see below). It is invoked by means of a function decorator:

```
@micropython.native
def foo(self, arg):
    buf = self.linebuf # Cached object
    # code
```

There are certain limitations in the current implementation of the native code emitter.

- Context managers are not supported (the `with` statement).
- Generators are not supported.
- If `raise` is used an argument must be supplied.

The trade-off for the improved performance (roughly twice as fast as bytecode) is an increase in compiled code size.

### 3.4.5 The Viper code emitter

The optimisations discussed above involve standards-compliant Python code. The Viper code emitter is not fully compliant. It supports special Viper native data types in pursuit of performance. Integer processing is non-compliant because it uses machine words: arithmetic on 32 bit hardware is performed modulo  $2^{**32}$ .

Like the Native emitter Viper produces machine instructions but further optimisations are performed, substantially increasing performance especially for integer arithmetic and bit manipulations. It is invoked using a decorator:

```
@micropython.viper
def foo(self, arg: int) -> int:
    # code
```

As the above fragment illustrates it is beneficial to use Python type hints to assist the Viper optimiser. Type hints provide information on the data types of arguments and of the return value; these are a standard Python language feature formally defined here [PEP0484](#). Viper supports its own set of types namely `int`, `uint` (unsigned integer), `ptr`, `ptr8`, `ptr16` and `ptr32`. The `ptrX` types are discussed below. Currently the `uint` type serves a single purpose: as a type hint for a function return value. If such a function returns `0xffffffff` Python will interpret the result as  $2^{**32} - 1$  rather than as `-1`.

In addition to the restrictions imposed by the native emitter the following constraints apply:

- Functions may have up to four arguments.
- Default argument values are not permitted.

- Floating point may be used but is not optimised.

Viper provides pointer types to assist the optimiser. These comprise

- `ptr` Pointer to an object.
- `ptr8` Points to a byte.
- `ptr16` Points to a 16 bit half-word.
- `ptr32` Points to a 32 bit machine word.

The concept of a pointer may be unfamiliar to Python programmers. It has similarities to a Python `memoryview` object in that it provides direct access to data stored in memory. Items are accessed using subscript notation, but slices are not supported: a pointer can return a single item only. Its purpose is to provide fast random access to data stored in contiguous memory locations - such as data stored in objects which support the buffer protocol, and memory-mapped peripheral registers in a microcontroller. It should be noted that programming using pointers is hazardous: bounds checking is not performed and the compiler does nothing to prevent buffer overrun errors.

Typical usage is to cache variables:

```
@micropython.viper
def foo(self, arg: int) -> int:
    buf = ptr8(self.linebuf) # self.linebuf is a bytearray object
    for x in range(20, 30):
        bar = buf[x] # Access a data item through the pointer
        # code omitted
```

In this instance the compiler “knows” that `buf` is the address of an array of bytes; it can emit code to rapidly compute the address of `buf[x]` at runtime. Where casts are used to convert objects to Viper native types these should be performed at the start of the function rather than in critical timing loops as the cast operation can take several microseconds. The rules for casting are as follows:

- Casting operators are currently: `int`, `bool`, `uint`, `ptr`, `ptr8`, `ptr16` and `ptr32`.
- The result of a cast will be a native Viper variable.
- Arguments to a cast can be a Python object or a native Viper variable.
- If argument is a native Viper variable, then cast is a no-op (i.e. costs nothing at runtime) that just changes the type (e.g. from `uint` to `ptr8`) so that you can then store/load using this pointer.
- If the argument is a Python object and the cast is `int` or `uint`, then the Python object must be of integral type and the value of that integral object is returned.
- The argument to a `bool` cast must be integral type (boolean or integer); when used as a return type the viper function will return `True` or `False` objects.
- If the argument is a Python object and the cast is `ptr`, `ptr8`, `ptr16` or `ptr32`, then the Python object must either have the buffer protocol with read-write capabilities (in which case a pointer to the start of the buffer is returned) or it must be of integral type (in which case the value of that integral object is returned).

The following example illustrates the use of a `ptr16` cast to toggle pin X1 `n` times:

```
BIT0 = const(1)
@micropython.viper
def toggle_n(n: int):
    odr = ptr16(stm.GPIOA + stm.GPIO_ODR)
    for _ in range(n):
        odr[0] ^= BIT0
```

A detailed technical description of the three code emitters may be found on Kickstarter here [Note 1](#) and here [Note 2](#)

## 3.5 MicroPython on Microcontrollers

MicroPython is designed to be capable of running on microcontrollers. These have hardware limitations which may be unfamiliar to programmers more familiar with conventional computers. In particular the amount of RAM and nonvolatile “disk” (flash memory) storage is limited. This tutorial offers ways to make the most of the limited resources. For instance, the Satlink 3 provides about 1MB of memory to Python programs, and the of 1MB of storage (pre-formatted).

### 3.5.1 RAM

When reducing RAM usage there are two phases to consider: compilation and execution. In addition to memory consumption, there is also an issue known as heap fragmentation. In general terms it is best to minimise the repeated creation and destruction of objects. The reason for this is covered in the section covering the *heap*.

#### Compilation Phase

When a module is imported, MicroPython compiles the code to bytecode which is then executed by the MicroPython virtual machine (VM). The bytecode is stored in RAM. The compiler itself requires RAM, but this becomes available for use when the compilation has completed.

If a number of modules have already been imported the situation can arise where there is insufficient RAM to run the compiler. In this case the import statement will produce a memory exception.

If a module instantiates global objects on import it will consume RAM at the time of import, which is then unavailable for the compiler to use on subsequent imports. In general it is best to avoid code which runs on import; a better approach is to have initialisation code which is run by the application after all modules have been imported. This maximises the RAM available to the compiler.

#### Execution Phase

There are a number of coding techniques for reducing RAM usage.

##### Constants

MicroPython provides a `const` keyword which may be used as follows:

```
from micropython import const
ROWS = const(33)
_COLS = const(0x10)
a = ROWS
b = _COLS
```

In both instances where the constant is assigned to a variable the compiler will avoid coding a lookup to the name of the constant by substituting its literal value. This saves bytecode and hence RAM. However the `ROWS` value will occupy at least two machine words, one each for the key and value in the globals dictionary. The presence in the dictionary is necessary because another module might import or use it. This RAM can be saved by prepending the name with an underscore as in `_COLS`: this symbol is not visible outside the module so will not occupy RAM.

The argument to `const()` may be anything which, at compile time, evaluates to an integer e.g. `0x100` or `1 << 8`. It can even include other `const` symbols that have already been defined, e.g. `1 << BIT`.

##### Needless object creation

There are a number of situations where objects may unwittingly be created and destroyed. This can reduce the usability of RAM through fragmentation. The following sections discuss instances of this.

### String concatenation

Consider the following code fragments which aim to produce constant strings:

```
var = "foo" + "bar"
var1 = "foo" "bar"
var2 = """\
foo\
bar"""
```

Each produces the same outcome, however the first needlessly creates two string objects at runtime, allocates more RAM for concatenation before producing the third. The others perform the concatenation at compile time which is more efficient, reducing fragmentation.

Where strings must be dynamically created before being fed to a stream such as a file it will save RAM if this is done in a piecemeal fashion. Rather than creating a large string object, create a substring and feed it to the stream before dealing with the next.

The best way to create dynamic strings is by means of the string `format` method:

```
var = "Temperature {:.2f} Pressure {:06d}\n".format(temp, press)
```

### Buffers

When accessing devices such as instances of `Serial`, using pre-allocated buffers avoids the creation of needless objects. Consider these two loops:

```
while True:
    var = serial.read(100)
    # process data

buf = bytearray(100)
while True:
    serial.readinto(buf)
    # process data in buf
```

The first creates a buffer on each pass whereas the second re-uses a pre-allocated buffer; this is both faster and more efficient in terms of memory fragmentation.

### Bytes are smaller than ints

On most platforms an integer consumes four bytes. Consider the two calls to the function `foo()`:

```
def foo(bar):
    for x in bar:
        print(x)
foo((1, 2, 0xff))
foo(b'\1\2\xff')
```

In the first call a tuple of integers is created in RAM. The second efficiently creates a `bytes` object consuming the minimum amount of RAM. If the module were frozen as bytecode, the `bytes` object would reside in flash.

### Strings Versus Bytes

Python3 introduced Unicode support. This introduced a distinction between a string and an array of bytes. MicroPython ensures that Unicode strings take no additional space so long as all characters in the string are ASCII (i.e. have a value < 126). If values in the full 8-bit range are required `bytes` and `bytearray` objects can be used to ensure that no additional space will be required. Note that most string methods (e.g. `strip()`) apply also to `bytes` instances so the process of eliminating Unicode can be painless.

```
s = 'the quick brown fox' # A string instance
b = b'the quick brown fox' # a bytes instance
```

Where it is necessary to convert between strings and bytes the string `encode` and the bytes `decode` methods can be used. Note that both strings and bytes are immutable. Any operation which takes as input such an object and produces another implies at least one RAM allocation to produce the result. In the second line below a new bytes object is allocated. This would also occur if `foo` were a string.

```
foo = b'   empty whitespace'
foo = foo.lstrip()
```

### Runtime compiler execution

The Python keywords `eval` and `exec` invoke the compiler at runtime, which requires significant amounts of RAM. Note that the `pickle` library employs `exec`. It may be more RAM efficient to use the `json` library for object serialisation.

## 3.5.2 The Heap

When a running program instantiates an object the necessary RAM is allocated from a fixed size pool known as the heap. When the object goes out of scope (in other words becomes inaccessible to code) the redundant object is known as “garbage”. A process known as “garbage collection” (GC) reclaims that memory, returning it to the free heap. This process runs automatically, however it can be invoked directly by issuing `gc.collect()`.

The discourse on this is somewhat involved. For a ‘quick fix’ issue the following periodically:

```
gc.collect()
gc.threshold(gc.mem_free() // 4 + gc.mem_alloc())
```

### Fragmentation

Say a program creates an object `foo`, then an object `bar`. Subsequently `foo` goes out of scope but `bar` remains. The RAM used by `foo` will be reclaimed by GC. However if `bar` was allocated to a higher address, the RAM reclaimed from `foo` will only be of use for objects no bigger than `foo`. In a complex or long running program the heap can become fragmented: despite there being a substantial amount of RAM available, there is insufficient contiguous space to allocate a particular object, and the program fails with a memory error.

The techniques outlined above aim to minimise this. Where large permanent buffers or other objects are required it is best to instantiate these early in the process of program execution before fragmentation can occur. Further improvements may be made by monitoring the state of the heap and by controlling GC; these are outlined below.

### Reporting

A number of library functions are available to report on memory allocation and to control GC. These are to be found in the `gc` and `micropython` modules. The following example may be pasted at the REPL (`ctrl e` to enter paste mode, `ctrl d` to run it).

```
import gc
import micropython
gc.collect()
micropython.mem_info()
print('-----')
print('Initial free: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
def func():
```

```

    a = bytearray(10000)
gc.collect()
print('Func definition: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
func()
print('Func run free: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
gc.collect()
print('Garbage collect free: {} allocated: {}'.format(gc.mem_free(), gc.mem_alloc()))
print('-----')
micropython.mem_info(1)

```

Methods employed above:

- `gc.collect()` Force a garbage collection. See footnote.
- `micropython.mem_info()` Print a summary of RAM utilisation.
- `gc.mem_free()` Return the free heap size in bytes.
- `gc.mem_alloc()` Return the number of bytes currently allocated.
- `micropython.mem_info(1)` Print a table of heap utilisation (detailed below).

The numbers produced are dependent on the platform, but it can be seen that declaring the function uses a small amount of RAM in the form of bytecode emitted by the compiler (the RAM used by the compiler has been reclaimed). Running the function uses over 10KiB, but on return `a` is garbage because it is out of scope and cannot be referenced. The final `gc.collect()` recovers that memory.

The final output produced by `micropython.mem_info(1)` will vary in detail but may be interpreted as follows:

Symbol	Meaning
.	free block
h	head block
=	tail block
m	marked head block
T	tuple
L	list
D	dict
F	float
B	byte code
M	module

Each letter represents a single block of memory, a block being 16 bytes. So each line of the heap dump represents 0x400 bytes or 1KiB of RAM.

### Control of Garbage Collection

A GC can be demanded at any time by issuing `gc.collect()`. It is advantageous to do this at intervals, firstly to pre-empt fragmentation and secondly for performance. A GC can take several milliseconds but is quicker when there is little work to do (about 1ms on the Pyboard). An explicit call can minimise that delay while ensuring it occurs at points in the program when it is acceptable.

Automatic GC is provoked under the following circumstances. When an attempt at allocation fails, a GC is performed and the allocation re-tried. Only if this fails is an exception raised. Secondly an automatic GC will be triggered if the amount of free RAM falls below a threshold. This threshold can be adapted as execution progresses:

```

gc.collect()
gc.threshold(gc.mem_free() // 4 + gc.mem_alloc())

```



This will provoke a GC when more than 25% of the currently free heap becomes occupied.

In general modules should instantiate data objects at runtime using constructors or other initialisation functions. The reason is that if this occurs on initialisation the compiler may be starved of RAM when subsequent modules are imported. If modules do instantiate data on import then `gc.collect()` issued after the import will ameliorate the problem.

### 3.5.3 String Operations

MicroPython handles strings in an efficient manner and understanding this can help in designing applications to run on microcontrollers. When a module is compiled, strings which occur multiple times are stored once only, a process known as string interning. In MicroPython an interned string is known as a `qstr`. In a module imported normally that single instance will be located in RAM, but as described above, in modules frozen as bytecode it will be located in flash.

String comparisons are also performed efficiently using hashing rather than character by character. The penalty for using strings rather than integers may hence be small both in terms of performance and RAM usage - a fact which may come as a surprise to C programmers.

### 3.5.4 Postscript

MicroPython passes, returns and (by default) copies objects by reference. A reference occupies a single machine word so these processes are efficient in RAM usage and speed.

Where variables are required whose size is neither a byte nor a machine word there are standard libraries which can assist in storing these efficiently and in performing conversions. See the `array`, `ustruct` and `uctypes` modules.

#### Footnote: `gc.collect()` return value

On Unix and Windows platforms the `gc.collect()` method returns an integer which signifies the number of distinct memory regions that were reclaimed in the collection (more precisely, the number of heads that were turned into frees). For efficiency reasons bare metal ports do not return this value.



## PYTHON AND MICROPYTHON RESOURCES

### 4.1 Tutorial for Xpert Basic Users

This is a quick primer, mainly for Xpert Basic users moving to MicroPython. Python is a clean, simple and elegant language which provides fairly direct equivalents to most BASIC statements, but for more advanced users also goes far beyond that. MicroPython is a subset of the Python language that tries to implement as much of Python 3.4 as will reasonably fit onto an Embedded Microprocessor. Over time, they've added some features from more recent versions of Python as well (at least where they found it appropriate).

Python makes an excellent language for writing general purpose applications and is a widely used, modern, and powerful language. For this primer we will focus on general programming constructs that would be familiar to Basic user's. For further reading, Python has a tutorial you can find here: <https://docs.python.org/3.4/tutorial/>, and more detailed documentation here: <https://docs.python.org/3.4/reference/index.html>. The homepage for the MicroPython project is located here: <https://micropython.org/>

#### 4.1.1 Interactive mode

In the early days of Basic, you could enter commands interactively. MicroPython too can be run interactively. The result of every expression is printed automatically. `>>>` indicates the python prompt. Pressing **tab** will even automatically complete commands, and **up-arrow** can be used to recall previous commands:

```
>>> a = 4
>>> a
4
```

```
>>> a + 1
5
```

#### 4.1.2 Getting started

Here's the familiar hello world program in Xpert Basic and in Python:

```
A=MsgBox ("Hello World")
```

becomes:

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

note that Python keywords are lowercase.

Assignment in Basic:

```
a = 4.5
b = 3
c = "Hello World"
```

is very much like assignment in Python:

```
a = 4.5
b = 3
c = "Hello World"
```

A Python variable can hold a value of any type, even nothing:

```
d = None
```

None is a special value that is different from zero or false or the empty string. This is useful in many situations. Consider, for example, a function which gets a number from the user and returns it. How should it indicate that the user has finished the list of numbers? Traditionally, a BASIC programmer might return zero or -1 to indicate this, but using None will prevent confusion if the user really does enter the reserved number.

Input is a function, not a statement, in Python:

```
number = input("Enter a number:")
```

### 4.1.3 WHILE LOOP

You can also enter multi-line statements, such as loops, at the prompt. The ‘...’ prompt indicates that it’s waiting for more input. Enter a blank line to complete the statement:

```
>>> x = 1
>>> while x < 10:
...     print x
...     x = x + 1
...
1
2
3
[etc]
```

BASIC programmers normally indent their code to show loops clearly. However, BASIC itself ignores the indentation and uses the **End Loop** statement to detect the end of a **Do While** loop. Python uses the indentation directly, so no end markers are needed. Python has a **break** statement that can be used to exit from a loop at any point similar to Basic’s **Exit Do** statement:

```
Finished = False
Do While Not finished
    REM Do stuff...
    If problem Then
        Exit Do
    End If
    REM More stuff...
End Loop
a = MsgBox("Done")
```

becomes:

```
finished = False
while not finished:
    # Do stuff...
    if problem:
        break
    # More stuff...
print "Done"
```

- Python uses # instead of REM for comments

#### 4.1.4 DO ... LOOP UNTIL

Python doesn't have DO UNTIL loops. These can normally be done directly as while loops, but if not, use break for the end condition. 'while True' is an infinite loop, as in BASIC:

```
Do
  REM Stuff
Loop Until finished
```

becomes:

```
while True:
    # Stuff
    if finished():
        break
```

#### 4.1.5 IF...THEN...ELSE...ENDIF

If statements are very similar. The use of indentation means that there is no difference between single line and multiline statements. Note the ':'; this starts all nested blocks. The 'else' must go at the start of a new line (this prevents any confusion as to which IF the ELSE goes with, as is possible in BASIC).

We can put more than one statement on a line in Python:

```
if x > 0 and x < 10: print("OK") else: print("Out of range!")
```

but isn't this easier to read:

```
if x > 0 and x < 10:
    print("OK")
else:
    print("Out of range!")
```

Like BASIC, Python considers 0 to be false, and any other integer to be true. Unlike BASIC, Python also allows non-integers:

```
>>> if "hello": print "yes"
...
yes
>>> if None: print "yes"
...

```

- An empty string is false, a non-empty one true.
- None is also considered to be false.

## 4.1.6 FUNCTION

Here's a simple function that doubles its input in Basic:

```
Function double(x)
    double = 2 * x
End Function
```

And here's the same in Python:

```
def double(x):
    return 2 * x
```

## 4.1.7 SUB

Here's a simple subroutine that says hello in Basic:

```
Sub hello
    A = MsgBox("Hello World")
End Sub

Call hello
```

And here's the same in Python:

```
>>> def hello():
...     print("Hello World")
>>> hello()
Hello World
```

There is no difference in Python between a function and a procedure. A procedure is just a function that doesn't return anything:

```
>>> a = hello()
Hello World
>>> print(a)
None
```

- To be more precise, a procedure is a function that returns the special value 'None'.
- You need the 'print' statement to print None; the interpreter skips it to save space.

## 4.1.8 ARRAY()

Python uses lists rather than arrays. They are rather more flexible (you can sort them, insert or delete entries, for example).

Xpert Basic allows any data type to be stored in an array:

```
a = Array("red", "green", "blue", 5.6)
```

and so do Python lists:

```
>>> a = ["red", "green", "blue"]
>>> a
['red', 'green', 'blue']
```

```

>>> a[0]
'red'
>>> a[1]
'green'
>>> a[2]
'blue'
>>> a[1] = "yellow"
>>> a[1]
'yellow'
>>> del a[1]
>>> a
['yellow', 'blue']

```

You can use them much like Xpert Basic arrays. You can even mix data types:

```

>>> a = ["red", "green", "blue"]
>>> a[1] = 4
>>> a
['red', 4, 'blue']

```

Negative indices count backwards from the end. This is very useful:

```

>>> a = ["red", "green", "blue"]
>>> a[1:3]
[4, 'blue']
>>> a[1:]
[4, 'blue']
>>> a[:1]
['red']
>>> a[:-1]
['red', 4]

```

- You can specify a start (inclusive) and end (exclusive) index to create a ‘slice’ of the original list. If you leave off one limit, it goes from the beginning or end.
- `[:-1]` means ‘all but the last element’, while `[1:]` means ‘all but the first’
- Python and Basic arrays both start with index 0

### 4.1.9 LEFT, RIGHT, MID

Python has none of these. Instead, you can treat a string as a list of characters and use the indexing and slicing notation as for lists:

```

>>> b = "Hello World"
>>> b[0]                                # Left(b, 1)
'H'
>>> b[1]                                # Mid(b, 2, 1)
'e'
>>> b[-1]                               # Right(b, 1)
'd'
>>> b[-2]                               # Mid(b, Len(b)-1, 1)
'l'
>>> b[1:]                               # Mid(b, 2)
'ello World'
>>> b[:-1]                              # Left(b, Len(b)-1)
'Hello Worl'

```

```
>>> b[2:4]                # Mid(b, 3, 2)
'll'
>>> b = b[:5] + " My" + b[5:] # Mid(b, 5) = " My"
>>> b
'Hello My World'
```

- Python strings start at index 0, Basic strings start at index 1.
- negative indices work from the end of the string, so b[-2] refers to the 2nd to last character in b

#### 4.1.10 FOR...NEXT

Python's for loops are completely different to BASIC's. Instead of providing start, end and step values, you provide a list to loop over.

For instance in Basic we'd iterate over each index in an array:

```
a = Array("red", "green", "blue")
For x = 0 to UBound(a)
    a=MsgBox("The next color is " + a[x])
Next x
```

whereas in Python we would just iterate over each item in an array:

```
>>> for x in ["red", "green", "blue"]:
...     print("The next color is ", x)
...
The next color is red
The next color is green
The next color is blue
```

You can get the BASIC behaviour by using the range() function to create a list of numbers to loop over, but you'll find you almost never need to use this.

For instance, to count the commas in a string, we might write the following in Basic:

```
a = "Hello,Bye,Fred"
count = 0
For c = 1 TO Len(a)
    If Mid(a, c, 1) = "," Then count = count + 1
Next c
a = MsgBox("String contains " + count + " commas")
```

becomes:

```
a = "Hello,Bye,Fred"
count = 0
for c in a:
    if c == ",": count += 1
print("String contains", count, "commas")
```

- **No need to loop over indices and then extract the element at that** index. Just loop over the list instead.
  - Use '==' for comparisons instead of '='. Python won't let you use '=' in an IF statement, so no risk of C-style errors.
  - += notation is the same
  - In fact, python provides a better way to count, as we shall see later.



### 4.1.11 ON ERROR GOTO... ERR

Python error handling is much simpler. For instance, here's a Basic function that needs to check for a division by zero error:

```
Const BE_DIVIDE_BY_ZERO = 25
Function Reciprocal(number)
  On Error Goto 0
  Reciprocal = 1.0/number
  If Err = BE_DIVIDE_BY_ZERO Then
    Reciprocal = 0.0
  End If
  On Error Resume Next
End Function
```

becomes:

```
def Reciprocal(number)
    try:
        answer = 1.0/number
    except ZeroDivisionError:
        answer = 0.0
```

- Any error in a 'try' block jumps to the 'except' clause.
- You can provide multiple except clauses for different types of errors.

If you don't provide a default handler ('except:') then the error is passed up to the enclosing 'try' block (possibly in a calling function) until it is either handled or there are no more try blocks. In that case, Python prints out the error, giving not only the line number of the problem, but showing the call in each function leading to it.

In fact, you should almost never use **except** on its own since unexpected errors should be reported with all their details. Only try to catch errors you want to handle specially (such as division by zero above).

### 4.1.12 OBJECTS

Time to come clean... Python is an object oriented language. This is a good thing, but requires a little adjustment to appreciate fully. Now, consider variables in BASIC. We think of a variable as a box with a label (the variable's name) and a value inside it. In an OO (Object Oriented) language, the values exist outside of any boxes on their own. Variables merely 'point to' values. The difference is, two variables can point to the same value. Consider:

```
>>> a = [1,2,3]
>>> b = [1,2,3]
>>> c = b
```

The final result can be visualized like this:

```
a ----> [1,2,3]
b ----> [1,2,3] <---- c
```

That is, we have created two lists. 'a' points to the first, while 'b' and 'c' point to the second. The '==' test in python checks if two arrays contain equal values, while the 'is' test checks if 'two' values are actually one:

```
>>> a == b
True
>>> b == c
True
```

```
>>> a is b
False
>>> b is c
True
```

The difference is important when you change something:

```
>>> b[1] = 42
>>> a
[1, 2, 3]
>>> b
[1, 42, 3]
>>> c
[1, 42, 3]
```

On the other hand, if you assign something to 'b' (rather than modifying the thing 'b' points to), you make 'b' point at that instead:

```
>>> b = [4, 5, 6]
a ----> [1, 2, 3]
c ----> [1, 42, 3]
b ----> [4, 5, 6]
```

You might be a bit worried about this:

```
>>> a = 1
>>> b = a
>>> a += 1
>>> b
[ what goes here? ]
```

Don't worry. After assigning to 'b', 'a' and 'b' do indeed point at the same object (the number 1):

```
a ----> 1 <---- b
```

However, incrementing 'a' has the effect of creating a new number, and making 'a' point at that:

```
a ----> 2
b ----> 1
```

You cannot change the number which 'a' points to in-place (numbers are thus said to be 'immutable'). Strings are also immutable. This means you can pass lists (arrays) to functions very quickly (because you're just making another pointer to the same array). If you want to copy a list, use the slicing notation to copy the whole thing:

```
>>> a = [1, 2, 3]
>>> b = a[:]
a ----> [1, 2, 3]
b ----> [1, 2, 3]
```

### 4.1.13 GARBAGE COLLECTION

OK, so we can create lots of new objects. How do we get rid of them again? The answer is simple: when it's no longer possible to refer to an object, Python will free it for you:

```
>>> a = [1, 2, 3]
>>> a = None
```

The first line creates three number objects and a list, and makes a pointer to the list:

```
a ----> [1, 2, 3]
```

The second line makes a pointer to the None object:

```
a ----> None
```

Since there's no way to get to the list or the objects it contains, they will all be freed when garbage collection occurs.

If we wanted to eliminate 'a' completely such that even the name no longer exists, we can delete it:

```
>>> del a
>>> a
NameError: name 'a' is not defined
```

We can even force garbage collection to occur:

```
import gc
gc.collect()
```

#### 4.1.14 Objects and methods

Because Basic has only a few types, it tends to represent everything else with numbers. File handles are an example; window handles another. Python tends to create new types for everything.

We have already seen a few types. You can use a type like a function, to create a new object of that type. Eg:

```
>>> a = int("4")
>>> a
4
>>> b = float(4)
>>> b
4.0
>>> c = str(4)
>>> c
"4"
>>> d = list()
>>> d
[]
```

Basic has a global selection of functions, and the names must be selected carefully to avoid clashes. Consider:

```
account = Account_New()
Call Account_Credit(account, 100)
a=MsgBox("Now has" + Account_check(account))
door = Door_New()
Call Door_Close(door)
Call Account_Close(account)
```

Because Python keeps track of types (and allows new ones to be created), it can automatically work out which of several functions to use, by keeping the functions 'inside the type'. Such functions are called 'methods':

```
account = Account()
account.Credit(100)
print("Now has", account.check())
door = Door()
```

```
door.Close()
account.Close()
```

- Here ‘Door’ and ‘Account’ are types (like ‘int’ or ‘str’), while ‘account’ and ‘door’ are values of that type (like 4 or “Hello”). Notice how we use the type to create new objects (‘instances’) of that type.
- Python knows that ‘door’ is a Door, so ‘door.Close()’ calls the close function for doors, whereas ‘account.Close()’ calls the close function for accounts.

Most objects in Python have many methods. Here are a few for strings:

```
>>> a = "Hello"
>>> a.upper()
"HELLO"
>>> a.lower()
"hello"
>>> " spaces ".strip()
"spaces"
>>> a = "Hello,Bye,Fred"
>>> print("String contains", a.count(", "), "commas")
String contains 2 commas
```

- Strings are immutable, so the above methods return a **new** string, leaving the original alone. In mutable objects, such as lists, some methods will modify the object in place.

While we’re on the subject of strings, I should point out that you can use either single or double quotes, as desired:

```
>>> a = "Hello"
>>> a = 'Hello'
>>> a = "Hello", she said.'
>>> a = '''In python, " and ' can be used to quote strings'''
```

- **Triple quotes of either type can also be used, and these can span** multiple lines.
- Strings can be any length; you can read an entire file into a string if you want.

### 4.1.15 SELECT ... CASE

There is no SELECT CASE statement. You can use **if** for a direct Basic translation:

```
if x == 1:
    print("One")
elif x == 2:
    print("Two")
elif x == 3:
    print("Three")
else:
    print("Many")
```

However, as with FOR loops, you’ll generally find that Python makes such things unnecessary. Consider:

```
For creature_number = 0 TO number_of_creatures - 1
  Select Case TypeOf[creature_number]
    Case Elf Move_Elf(creature_number)
    Case Goblin Move_Goblin(creature_number)
    Case Dragon Move_Dragon(creature_number)
    ...
```

```
End Select
Next creature_number
```

might be written in Python as:

```
for creature in creatures:
    creature.Move()
```

As long as we create a new type for each kind of creature, Python will call the correct ‘move’ function for each creature automatically!

### 4.1.16 USER ERRORS

It’s not possible to create your own error in Xpert Basic, but in Python exceptions are not limited to just the system:

```
def UserFunction():
    raise Exception("User error!")
```

Exception is a type, so this creates a new Exception object and raises it as an error. It can be caught in a try block, as shown below:

```
try:
    UserFunction()
except Exception:
    print("failed")
```

When you know how to create your own types, you can use that to create different types of error (any object can be raised as an error, even a string, but there are conventions...).

### 4.1.17 OPEN FILES

This:

```
f = FreeFile
Open "MyFile" For Input As f
```

becomes:

```
>>> f = open("MyFile")
or
```

```
>>> f = open("MyFile", "r")
```

- a file is a type, like ‘str’ or ‘int’
- ‘r’ means ‘for reading’

You can use the file type’s methods to read from it:

```
>>> f.readline()
'The first line\r\n'
>>> f.readline()
'The second line\r\n'
>>> f.readline()
'The end\r\n'
```

```
>>> f.readline()
''
```

Each line read includes the newline character (`'\n'`) and possibly a carriage return (`'\r'`), so you can tell the difference between a blank line (`'\n'` or `'\r\n'`) and the end of the file (`''`). More commonly, you'll use a file object as the sequence in a list. In this case, the loop iterates over the lines:

```
my_data = open('MyFile')
for line in my_data:
    print("Next line is", line)
my_data.close()
```

Or, indeed:

```
for line in open('MyFile'):
    print("Next line is", line)
```

- When the loop finishes, the file object is unreachable (since we never assigned it to a variable). Therefore, the garbage collector frees it, closing the file for us automatically.
- Earlier, I said that for loops take a list to work on. In fact, a for loop takes an *iterator* (anything that can generate a sequence of values). Therefore, lists, files and even user-defined types can be used in for loops.

For writing, pass `'w'` to the constructor rather than `'r'` (the type used to create a new object):

```
f = FreeFile
Open "Report.txt" For Output As f
Print f, "Hello World"
Close f
```

becomes:

```
f = open("Report.txt", 'w')
f.write("Hello World\r\n")
f.close()
```

or:

```
f = open("Report.txt", 'w')
print("Hello World\r", file=f)
f.close()
```

When performing file I/O with `print` it will automatically append the newline character (`'\n'`) to the output, but if the file is intended to be read by a Microsoft Windows system, you'll need to add a carriage return (`'\r'`) character as well as shown in the example. Unix and Linux based systems use just the newline character (`'\n'`) to mark the end of lines.

### 4.1.18 LIBRARIES

Libraries (or modules) contain functions we can use:

```
>>> import math
>>> math.cos(0)
1.0
```

- `import` loads routines from another file (called a 'module')
- You access a module's functions using the same notation as an object's methods. This avoids name conflicts.

For example, most math functions exist in both normal and ‘complex’ forms:

```
>>> import cmath, math
>>> math.cos(0)
1.0
>>> cmath.cos(0)
(1-0j)
```

You can also import functions directly:

```
>>> from math import cos, sin
>>> cos(0)
1.0
>>> sin(0)
0.0
```

You can use import to split your own programs into several files. Name the files with a ‘.py’ extension, and use import without the extension to load the file.

#### 4.1.19 HELP

Use the help function to find out about anything in Python. You can get help on functions, types and modules!

```
>>> import math
>>> help(math)
>>> help(math.cos)
>>> help(str)
```

- MicroPython’s help system provides primarily names and types, whereas desktop Python’s help provides more thorough documentation.

Notice how we are passing apparently abstract ideas as function arguments! In fact, all these things are real Python values, and can be assigned just like anything else:

```
>>> import math
>>> print(math.cos(0))
1.0
>>> print(math.cos)
<function>
>>> print math
<module 'math'>
>>> a = math.cos
>>> a(0)
1.0
```

#### 4.1.20 AND, OR, XOR, ^

Most python operators are the same as in BASIC. The bit-wise operators are different, however. XOR is written ^ in python, AND as & and OR as | (and ^ becomes \*\*):

```
>>> 6 & 3 # (6 And 3)
2
>>> 6 | 3 # (6 Or 3)
7
>>> 6 ^ 3 # (6 Xor 3)
```

```
5
>>> 6 ** 3 # (6 ^ 3)
216
>>> 6 ** 333 # (??? Basic can't do this ???)
1331577...4415616
```

- Basic integers are limited in range to roughly +/- 2 billion, but this is not the case for MicroPython which can handle arbitrarily large integers.

BASIC also uses the bitwise operators AND and OR for boolean operations. Use the normal python ‘and’ and ‘or’ operators for this. These are ‘short-circuit’ operators; they only evaluate as far as they need to to get the result:

```
>>> 0 and 1/0
0
```

- There is no divide by zero error, because Python only evaluates as far as the 0 before realizing the result is going to be false.

### 4.1.21 STATIC AND DIM

Python doesn’t have STATIC or DIM; any variable assigned in a function is automatically considered to be a local variable, and any variable defined outside of a function is considered to be a global variable. The keyword ‘global’ must be used to indicate that a variable is NOT local:

```
>>> def inc_a():
...     global a
...     a += 1
...
>>> a = 1
>>> inc_a()
>>> print a
2
```

- When you use a variable without assigning to it, Python uses the variable assigned to in the closest enclosing lexical scope.

### 4.1.22 PASSING BY REFERENCE

Xpert Basic supports “passing by reference” which means if you pass a variable to a subroutine, and the subroutine changes the variable, then the variable passed in will change too. So, for instance in the example below we create a function that will increment a variable as long as it’s less than 10:

```
Function Increment(test)
  If test < 10 Then
    test = test + 1
    Increment = True
  Else
    Increment = False
  End If
End Function

a = 2

If Increment(a) Then
```



```
' a will now be 3
End If
```

Python, however, passes variables by value. If the variable is a list, we can change the list, or add an item to the list, but any change to the variable itself will not affect the original variable. So, how can we get around this? How can we implement an Increment function? Do we have to put our variable in a list? Fortunately, not. The key is to take advantage of the fact that Python can return multiple values like we do in this example:

```
def Increment(test):
    if test < 10:
        return True, test+1
    else:
        return False, test

a = 2

ok, a = Increment(a)
if ok:
    ' a will now be 3
```

Why would the Python designers not implement a feature that can be so useful? The reason is because passing by reference has become frowned upon due to how easy it is to introduce an obscure bug in a program if the programmer forgets about the “side effects” of a subroutine or function. Fortunately, Python doesn’t just leave us with this lofty goal without also giving us tools to help realize them.

Still not convinced? Here’s an example where “passing by reference” can cause a bug:

```
Sub PumpOn(channel)
    ' pulse the pump on channel
    Digital 1, channel, True
    Sleep 0.1
    Digital 1, channel, False
End Sub

Sub PumpOff(channel)
    ' the pump off channel always comes after the pump on channel
    channel = channel + 1
    Digital 1, channel, True
    Sleep 0.1
    Digital 1, channel, False
End Sub

' Pulse the pump on channel 2 ten times:
For i = 1 To 10
    Call PumpOn(2)
    Sleep 5
    Call PumpOff(2)
Next i
```

What’s the problem? Well, nothing yet, but there is a bug sleeping in the code. If we decided later we didn’t want to hardcode the pump channel as 2 and switched to using a variable instead, then each call to PumpOff would increment the new variable by side-effect and the code would no longer work.

### 4.1.23 Future reading

A few other interesting bits to look out for in Python:

- The `%` operator can be used with a string to insert formatted values. Eg:

```
>>> "Pi is %.3f (to three decimal places)" % math.pi
'Pi is 3.142 (to three decimal places)'
```

- The `'class'` keyword lets you create your own types. Vital to take advantage of python properly:

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> p1 = Point(3, 4)
>>> p2 = Point(7, 8)
>>> p1.x
3
>>> p2.y
8
```

- There is a very useful `dict` type. It holds a set of elements, each with a `key` and a `value` and can look up a value from a key in constant time (no matter how big the dictionary gets). Very useful.
- The `assert` statement checks that something is true, and raises an exception if not. Sprinkle generously around your code to check for programming errors.
- Everything is an object! Functions, modules, classes can all be assigned to variables just like other values. You can pass a function to another function, or get one back. Consider this:

```
>>> def double(x): return 2 * x
...
>>> map(double, [1,2,3])
[2, 4, 6]
```

The `map` function applies a function to every item in a list. In this case we doubled every value in the list `[1,2,3]`.

### 4.1.24 Credits

Contents inspired by and adapted from: <http://rox.sourceforge.net/desktop/book/export/html/44.html>

## 4.2 Python Tutorials:

- *Tutorial for Xpert Basic Users*
- A Byte of Python, by Swaroop C.H., is an introductory text for people with no previous programming experience
- Official Python Documentation
- Official Python Tutorial
- Differences between MicroPython and Python
- Nearly standalone Win32 version of Python 3.5 (just need `python.exe` and `python35.dll`) is at `K:\Engshare\Satlink3\Python`
- Google's Python Style Guide

## 4.3 MicroPython Resources:

- Home page
- Discussion Forums
- Source code
- MicroPython Tutorial (using their PyBoard)
- Library Documentation
- Original KickStarter for MicroPython
- Follow on KickStarter to support the ESP8266 (WiFi IoT board)
- KickStarter for LoPy board (LoRa, WiFi, BLE IoT board)
- KickStarter for FiPy board (LoRa, WiFi, BLE, LTE, SigFox IoT board)
- MicroPython Tutorial with the BBC:MicroBit
- Standalone Win32 version of MicroPython.exe 1.8.7 is at K:\Engshare\Satlink3\Python
- Documentation for MicroPython 1.8.7 in a PDF (link to Original)

## 4.4 Python Videos:

- Learn Python in One Video by Derek Banas. Rapid overview of python syntax and manipulation of strings, lists, tuples, dictionary, conditionals, loops, functions, user input, string functions, file I/O, classes, ect.:
- Damien George introduces MicroPython and why he created it:
- Multi-part Video on Python starting with the very basics:
- Learn about Decorators and when to use them:
- Creating beautiful python code. Comparing typical programmer approach with code examples to equivalent python implementation. A talk given by one of the python developers:
- PyCharm Overview:

## 4.5 More Python Articles and Links

- Python Resources for Hydrologists
- Fast Fourier Transform for uPy (uses ARM Thumb integer math)

## 4.6 MicroPython Built-In Libraries

A number of libraries have been written for MicroPython, these are built-in to a port and can be imported by Python programs. The SL3 port of MicroPython initially includes all the libraries that could be easily ported and did not require additional hardware or conflict with the SL3's use of hardware. Some of the libraries are bare bones ports, and most are optional.

- Nice documentation of libraries available for the pyBoard port of MicroPython: <http://docs.micropython.org/en/v1.8.3/pyboard/library/index.html>

Here's a list of the standard libraries that have been ported to the SL3:

- array
- builtins
- cmath
- collections
- gc
- math
- micropython
- struct
- sys
- errno
- \_thread

Here's a list of the optional libraries that have been ported to the SL3 (which ones we keep is tbd):

- umachine
- pyb
- stm
- binascii
- re
- zlib
- json
- heapq
- hashlib
- time
- struct
- machine
- urandom
- ctypes
- os

Currently not supported:

- select
- lwip
- framebuffer
- btree
- axtls
- mbedtls
- timeq

- webrepl
- websocket
- dupterm
- vfs

## 4.7 MicroPython Standard Libraries

MicroPython takes a minimalistic approach but yet tries to support as much of Python 3.5/3.6 as possible. To do this any standard Python libraries which can be written in Python are written in Python. It looks very impressive, but many of the libraries are just stubs. For instance if you look at ‘poplib’ and examine ‘poplib.py’ the last remark for it was “poplib: Add dummy module” indicating that the last update only created a “dummy module” or a stub for where code should go in the future. It’s also possible to embed precompiled Python code into a port using a concept called “frozen modules”. The future usefulness of these libraries is up to the community.

- This is a link to the library repository: <https://github.com/micropython/micropython-lib>

## 4.8 MicroPython Cross Compiler and Windows Version.

Roberthh has a web page with a pre-built copy of MicroPython built for Windows, as well as the mpy-cross Cross Compiler that converts .py in to .mpy. The latest version of mpy-cross is not compatible with the version of uPy the SL3 is currently using, but we can access the history, and the Oct 11, 2016 version seems to work. MicroPython.exe is based on the Linux port, and will have differences from our SL3 port.

see here: <https://github.com/robert-hh/Shared-Stuff>

## 4.9 Udacity Free Courses

Programming Foundations with Python: (6 weeks, beginner level)

<https://www.udacity.com/course/programming-foundations-with-python-ud036>

Very beginner level but using Python: (3 Months, beginner level)

<https://www.udacity.com/course/intro-to-computer-science-cs101>



## MICROPYTHON LICENSE INFORMATION

The MIT License (MIT)

Copyright (c) 2013-2015 Damien P. George, and others

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.





## INDICES AND TABLES

- genindex
- modindex
- search



## PYTHON MODULE INDEX

—  
\_thread, ??

**a**  
array, ??

**b**  
builtins, ??

**c**  
cmath, ??

**g**  
gc, ??

**m**  
machine, 16  
math, ??  
micropython, 16

**s**  
serial, 19  
sl3, 24  
sys, ??

**u**  
ubinascii, ??  
ucollections, ??  
uctypes, 16  
uerrno, ??  
uhashlib, ??  
uheapq, ??  
uio, ??  
ujson, ??  
uos, ??  
ure, ??  
ustruct, ??  
utime, ??