Micropython usermod documentation

Release 1.32

Zoltán Vörös

Oct 12, 2019

Content:

CHAPTER 1

Introduction

So, you have somehow bumped into micropython, fallen in love with it in an instance, broken your piggy bank, and run off, head over heels, to order a pyboard. You have probably paid extra for the expedited shipping. Once the pyboard arrived, you became excited like a puppy with a bone. You played with the hardware, learnt how to access the accelerometer, switch, LEDs, and temperature sensor, and you successfully communicated with other devices via the I2C, SPI, USART, or CAN interfaces. You have plugged the board in a computer, and driven someone crazy by emulating a seemingly disoriented mouse on it. You have even tried to divide by zero, just to see if the chip would go up in flames (this was vicious, by the way), and noticed that the interpreter smartly prevented such things from happening. You have written your own python functions, even compiled them into frozen modules, and burnt the whole damn thing onto the microcontroller. Then you have toyed with the on-board assembler, because you hoped that you could gain some astronomical factors in speed. (But you couldn't.)

And yet, after all this, you feel somewhat unsatisfied. You find that you want to access the periphery in a special way, or you need some fancy function that, when implemented in python itself, seems to consume too much RAM, and takes an eternity to execute, and assembly, with its limitations, is just far too awkward for it. Or perhaps, you are simply dead against making your code easily readable by writing everything in python, and you want to hide the magic, just for the heck of it. But you still want to retain the elegance of python.

If, after thorough introspection and soul-searching, you have discovered these latter symptoms in yourself, you have two choices: either you despair, scrap your idea, and move on, or you learn how the heavy lifting behind the micropython facade is done, and spin your own functions, classes, and methods in C. As it turns out, it is not that hard, once you get the hang of it. The sole trick is to get the hang of it. And this is, where this document intends to play a role.

On the following pages, I would like to show how new functionality can be added and exposed to the python interpreter. I will try to discuss all aspects of micropython in an approachable way. Each concept will be presented in an implementation, stripped to the bare minimum, that you can compile right away, and try yourself. (The code here has been tested against micropython v.1.11.) At the end of each chapter, I will list the discussed code in its entirety, and I also include a link the the source, so that copying and pasting does not involve copious amounts of work. Moreover, I include a small demonstration, so that we can actually see that our code works. The code, as well as the source of this document are also available under https://github.com/v923z/micropython-usermod. The simples way of getting started is probably cloning the repository with

```
git clone https://github.com/v923z/micropython-usermod.git
```

As for the source: all that you see here originates from a single jupyter notebook. That's right, the documentation, the C

source, the compilation, and the testing. You can find the notebook at https://github.com/v923z/micropython-usermod/blob/master/docs/micropython-usermod.ipynb. And should you wonder, everything is under the MIT licence.

I start out with a very simple module and slowly build upon it. At the very end of the discussion, I will outline my version of a general-purpose math library, similar to numpy. In fact, it was when I was working on this math module that I realised that a decent programming guide to micropython is sorely missing, hence this document. Obviously, numpy is a gigantic library, and we are not going to implement all aspects of it. But we will be able to define efficiently stored arrays on which we can do vectorised computations, work with matrices, invert and contract them, fit polynomials to measurement data, and get the Fourier transform of an arbitrary sequence. I do hope that you find the agenda convincing enough!

One last comment: I believe, all examples in this document could be implemented with little effort in python itself, and I am definitely not advocating the inclusion of such trivial cases in the firmware. I chose these examples on two grounds: First, they are all simple, almost primitive, but for this very reason, they demonstrate a single idea without distraction. Second, having a piece of parallel python code is useful insofar as it tells us what to expect, and it also encourages us to implement the C version such that it results in *pythonic* functions.

1.1 Code blocks

You'll encounter various kinds of code blocks in this document. These have various scopes, which are listed here:

- if a code block begins with an exclamation mark, the content is meant to be executed on the command line.
- if the code block looks like a piece of python code, it should be run in a python interpreter.
- if the heading of the code block is `%%micropython`, then, well, you guessed it right, the content should be passed to the micropython interpreter of your port of choice.
- other code segments can be C code, or a makefile. These should be easy to recognise, because both of these have a header with a link to the location of the file.

CHAPTER 2

The micropython code base

Since we are going to test our code mainly on the unix port, we set that as the current working directory.

```
!cd ../../micropython/ports/unix/
```

The micropython codebase itself is set up a rather modular way. Provided you cloned the micropython repository with

```
!git clone https://github.com/micropython/micropython.git
```

onto your computer, and you look at the top-level directories, you will see something like this:

```
!ls ../../../micropython/
```

```
ACKNOWLEDGEMENTS    docs      lib    pic16bit   teensy   zephyr
bare-arm        drivers   LICENSE    py         tests
cc3200                  esp8266   logo       qemu-arm   tools
CODECONVENTIONS.md  examples  minimal        README.md  unix
CONTRIBUTING.md     extmod    mpy-cross  stmhal     windows
```

Out of all the directoties, at least two are of particular interest. Namely, `/py/`, where the python interpreter is implemented, and `/ports/`, which contains the hardware-specific files. All questions pertaining to programming micropython in C can be answered by browsing these two directories, and perusing the relevant files therein.

2.1 User modules in micropython

Beginning with the 1.10 version of micropython, it became quite simple to add a user-defined C module to the firmware. You simply drop two or three files in an arbitrary directory, and pass two compiler flags to `make` like so:

```
!make USER_C_MODULES=../../../user_modules CFLAGS_EXTRA=-DMODULE_EXAMPLE_ENABLED=1 all
```

Here, the `USER_C_MODULES` variable is the location (relative to the location of `make`) of your files, while `CFLAGS_EXTRA` defines the flag for your particular module. This is relevant, if you have many modules, but you want to include only some of them.

Alternatively, you can set the module flags in `mpconfigport.h` (to be found in the port's root folder, for which you are compiling) as

```
#define MODULE_SIMPLEFUNCTION_ENABLED (1)
#define MODULE_SIMPLECLASS_ENABLED (1)
#define MODULE_SPECIALCLASS_ENABLED (1)
#define MODULE_KEYWORDFUNCTION_ENABLED (1)
#define MODULE_CONSUMEITERABLE_ENABLED (1)
#define MODULE_VECTOR_ENABLED (1)
#define MODULE_RETURNITERABLE_ENABLED (1)
#define MODULE_PROFILING_ENABLED (1)
#define MODULE_MAKEITERABLE_ENABLED (1)
#define MODULE_SUBSCRIPTITERABLE_ENABLED (1)
#define MODULE_SLICEITERABLE_ENABLED (1)
#define MODULE_VARARG_ENABLED (1)
#define MODULE_STRINGARG_ENABLED (1)
```

and then call `make` without the `CFLAGS_EXTRA` flag:

```
!make USER_C_MODULES=../../../user_modules all
```

This separation of the user code from the micropython code base is definitely a convenience, because it is much easier to keep track of changes, and also because you can't possibly screw up micropython itself: you can also go back to a working piece of firmware by dropping the `USER_C_MODULES` argument of `make`.

CHAPTER 3

micropython internals

Before exploring the exciting problem of micropython function implementation in C, we should first understand how python objects are stored and treated at the firmware level.

3.1 Object representation

Whenever you write

```
>>> a = 1
>>> b = 2
>>> a + b
```

on the python console, first the two new variables, `a`, and `b` are created and a reference to them is stored in memory. Then the value of 1, and 2, respectively, will be associated with these variables. In the last line, when the sum is to be computed, the interpreter somehow has to figure out, how to decipher the values stored in `a`, and `b`: in the RAM, these two variables are just bytes, but depending on the type of the variable, different meanings will be associated with these bytes. Since the type cannot be known at compile time, there must be a mechanism for keeping stock of this extra piece of information. This is, where `mp_obj_t`, defined in `obj.h`, takes centre stage.

If you cast a cursory glance at any of the C functions that are exposed to the python interpreter, you will always see something like this

```
mp_obj_t some_function(mp_obj_t some_variable, ...) {
    // some_variable is converted to fundamental C types (bytes, ints, floats,␣
↪pointers, structures, etc.)
    ...
}
```

Variables of type `mp_obj_t` are passed to the function, and the function returns the results as an object of type `mp_obj_t`. So, what is all this fuss this about? Basically, `mp_obj_t` is nothing but an 8-byte segment of the memory, where all concrete objects are encoded. There can be various object encodings. E.g., in the `A` encoding, integers are those objects, whose rightmost bit in this 8-byte representation is set to 1, and the value of the integer can then be retrieved by shifting these 8 bytes by one to the right, and then applying a mask. In the `B` encoding, the

variable is an integer, if its value is 1, when ANDed with 3, and the value will be returned, if we shift the 8 bytes by two to the right.

3.2 Type checking

Fortunately, we do not have to be concerned with the representations and the shifts, because there are pre-defined macros for such operations. So, if we want to find out, whether `some_variable` is an integer, we can inspect the value of the Boolean

```
MP_OBJ_IS_SMALL_INT(some_variable)
```

The integer value stored in `some_variable` can then be gotten by calling `MP_OBJ_SMALL_INT_VALUE`:

```
int value_of_some_variable = MP_OBJ_SMALL_INT_VALUE(some_variable);
```

These decoding steps take place somewhere in the body of `some_function`, before we start working with native C types. Once we are done with the calculations, we have to return an `mp_obj_t`, so that the interpreter can handle the results (e.g., show it on the console, or pipe it to the next instruction). In this case, the encoding is done by calling

```
mp_obj_new_int(value_of_some_variable)
```

More generic types can be treated with the macro `MP_OBJ_IS_TYPE`, which takes the object as the first, and a pointer to the type as the second argument. Now, if you want to find out, whether `some_variable` is a tuple, you could apply the `MP_OBJ_IS_TYPE` macro,

```
MP_OBJ_IS_TYPE(some_variable, &mp_type_tuple)
```

While the available types can be found in `obj.h`, they all follow the `mp_type_` + python type pattern, so in most cases, it is not even necessary to look them up. We should also note that it is also possible to define new types. When done properly, `MP_OBJ_IS_TYPE` can be called on objects with this new type, i.e.,

```
MP_OBJ_IS_TYPE(myobject, &my_type)
```

will just work. We return to this question later.

3.3 python constants

At this point, we should mention that python constants,`True` (in C `mp_const_true`), `False` (in C `mp_const_false`), `None` (in C `mp_const_none`) and the like are also defined in `obj.h`. These are objects of type `mp_obj_t`, as almost anything else, so you can return them from a function, when the function is meant to return directly to the interpreter.

CHAPTER 4

Developing your first module

Having seen, what the python objects look like to the interpreter, we can start with our explorations in earnest. We begin by adding a simple module to micropython. The module will have a single function that takes two numbers, and adds them. I know that this is the most exciting thing since sliced bread, and you have always wondered, why there isn't a built-in python function for such a fascinating task. Well, wonder no more! From this moment, *your* micropython will have one.

First I show the file in its entirety (20 something lines all in all), and then discuss the parts.

https://github.com/v923z/micropython-usermod/tree/master/snippets/simplefunction/simplefunction.c

```
#include "py/obj.h"
#include "py/runtime.h"

STATIC mp_obj_t simplefunction_add_ints(mp_obj_t a_obj, mp_obj_t b_obj) {
    int a = mp_obj_get_int(a_obj);
    int b = mp_obj_get_int(b_obj);
    return mp_obj_new_int(a + b);
}

STATIC MP_DEFINE_CONST_FUN_OBJ_2(simplefunction_add_ints_obj, simplefunction_add_
↪ints);

STATIC const mp_rom_map_elem_t simplefunction_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_simplefunction) },
    { MP_ROM_QSTR(MP_QSTR_add_ints), MP_ROM_PTR(&simplefunction_add_ints_obj) },
};
STATIC MP_DEFINE_CONST_DICT(simplefunction_module_globals, simplefunction_module_
↪globals_table);

const mp_obj_module_t simplefunction_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&simplefunction_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_simplefunction, simplefunction_user_cmodule, MODULE_
↪SIMPLEFUNCTION_ENABLED);
```

(continues on next page)

(continued from previous page)

4.1 Header files

A module will not be too useful without at least two includes: `py/obj.h`, where all the relevant constants and macros are defined, and `py/runtime.h`, which contains the declaration of the interpreter. Many a time you will also need `py/builtin.h`, where the python built-in functions and modules are declared.

4.2 Defining user functions

After including the necessary headers, we define the function that is going to do the heavy lifting. By passing variables of `mp_obj_t` type, we make sure that the function will be able to accept values from the python console. If you happen to have an internal helper function in your module that is not exposed in python, you can pass whatever type you need. Similarly, by returning an object of `mp_obj_t` type, we make the results visible to the interpreter, i.e., we can assign the value returned to variables.

The downside of passing `mp_obj_t`s around is that you cannot simply assign them to usual C variables, i.e., when you want to operate on them, you have to extract the values first. This is why we have to invoke the `mp_obj_get_int()` function, and conversely, before returning the results, we have to do a type conversion to `mp_obj_t` by calling `mp_obj_new_int()`. These are the decoding/encoding steps that we discussed above.

4.2.1 Referring to user functions

We have now a function that should be sort of OK (there is no error checking whatsoever, so you are at the mercy of the firmware, when, e.g., you try to pass a float to the function), but the python interpreter still cannot work with. For that, we have to turn our function into a function object. This is what happens in the line

```
STATIC MP_DEFINE_CONST_FUN_OBJ_2(simplefunction_add_ints_obj, simplefunction_add_
↪ints);
```

The first argument of the macro is the name of the function object to which our actual function, the last argument, will be bound. Now, these `MP_DEFINE_CONST_FUN_OBJ_*` macros, defined in the header file `py/obj.h` (one more reason not to forget about `py/obj.h`), come in seven flavours, depending on what kind of, and how many arguments the function is supposed to take. In the example above, our function is meant to take two arguments, hence the 2 at the end of the macro name. Functions with 0 to 4 arguments can be bound in this way.

But what, if you want a function with more than four arguments, as is the case many a time in python? Under such circumstances, one can make use of the

```
STATIC MP_DEFINE_CONST_FUN_OBJ_VAR(obj_name, n_args_min, fun_name);
```

macro, where the second argument, an integer, gives the minimum number of arguments. The number of arguments can be bound from above by wrapping the function with

```
STATIC MP_DEFINE_CONST_FUN_OBJ_VAR_BETWEEN(obj_name, n_args_min, n_args_max, fun_
↪name);
```

Later we will see, how we can define functions that can also take keyword arguments.

At this point, we are more or less done with the C implementation of our function, but we still have to expose it. This we do by adding a table, an array of key/value pairs to the globals of our module, and bind the table to the

_module_globals variable by applying the MP_DEFINE_CONST_DICT macro. This table should have at least one entry, the name of the module, which is going to be stored in the string MP_QSTR___name__.

These MP_QSRT_ items are the C representation of the python strings that come at the end of them. So, MP_QSRT_foo_bar in C will be turned into a name, foo_bar, in python. foo_bar can be a constant, a function, a class, a type, etc., and depending on what is associated with it, different things will happen on the console, when foo_bar is invoked. But the crucial point is that, if you want foo_bar to have any meaning in python, then somewhere in your C code, you have to define MP_QSRT_foo_bar.

The second key-value pair of the table is the pointer to the function that we have just implemented, and the name that we want to call the functions in python itself. So, in the example below, our simplefunction_add_ints function will be invoked, when we call add_ints in the console.

```
STATIC const mp_rom_map_elem_t simplefunction_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_simplefunction) },
    { MP_ROM_QSTR(MP_QSTR_add_ints), MP_ROM_PTR(&simplefunction_add_ints_obj) },
};
STATIC MP_DEFINE_CONST_DICT(simplefunction_module_globals, simplefunction_module_
↪globals_table);
```

This three-step pattern is common to all function implementations, so I repeat it here:

1. implement the function
2. then turn it into a function object (i.e., call the relevant form of MP_DEFINE_CONST_FUN_OBJ_*)
3. and finally, register the function in the name space of the module (i.e., add it to the module's globals table, and turn the table into a dictionary by applying MP_DEFINE_CONST_DICT)

It doesn't matter, whether our function takes positional arguments, or keyword argument, or both, these are the required steps.

Having defined the function object, we have finally to register the module with

```
MP_REGISTER_MODULE(MP_QSTR_simplefunction, simplefunction_user_cmodule, MODULE_
↪SIMPLEFUNCTION_ENABLED);
```

This last line is particularly useful, because by setting the MODULE_SIMPLEFUNCTION_ENABLED variable in mpconfigport.h, you can selectively exclude modules from the linking, i.e., if in mpconfigport.h, which should be in the root directory of the port you want to compile for,

```
#define MODULE_SIMPLEFUNCTION_ENABLED (1)
```

then simplefunction will be included in the firmware, while with

```
#define MODULE_SIMPLEFUNCTION_ENABLED (0)
```

the module will be dropped, even though the source is in your modules folder. (N.B.: the module will still be compiled, but not linked.)

4.3 Compiling our module

The implementation is done, and we would certainly like to see some results. First we generate a makefile, which will be inserted in the module's own directory, simplefunction/.

https://github.com/v923z/micropython-usermod/tree/master/snippets/simplefunction/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/simplefunction.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

If `mpconfigport.h` is augmented with

```
#define MODULE_SIMPLEFUNCTION_ENABLED (1)
```

you should be able to compile the module above by calling

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

We can then test the module as

```
%%micropython

import simplefunction
print(simplefunction.add_ints(123, 456))
```

```
579
```

What a surprise! It works! It works!

4.4 Compiling for the microcontroller

As pointed out at the very beginning, our first module was compiled for the unix port, and that it, why we set `../../micropython/ports/unix/` as our working directory. In case, we would like to compile for the microcontroller, we would have to modify the `mpconfigport.h` file of the port (e.g., in `micropython/ports/stm32/`) as shown in Section *User modules*.

Next, in the compilation command, one has to specify the target board, e.g., pyboard, version 1.1, and probably the path to the cross-compiler, if that could not be installed system-wide. You would issue the make command in the directory of the port, e.g., `micropython/ports/stm32/`, and the path in the `CROSS_COMPILE` argument must be either absolute, or given relative to `micropython/ports/stm32/`.

```
make BOARD=PYBV11 CROSS_COMPILE=<Path where you uncompressed the toolchain>/bin/arm-
↪none-eabi-
```

You will find your firmware under `micropython/ports/stm32/build-PYBV11/firmware.dfu`, and you can upload it by issuing

```
!python ../../tools/pydfu.py -u build-PYBV11/firmware.dfu
```

on the command line. More detailed explanation can be found under https://github.com/micropython/micropython/wiki/Pyboard-Firmware-Update.

CHAPTER 5

Error handling

There will be cases, when something goes wrong, and you want to bail out in an elegant way. If bailing out, and elegance can be used in the same sentence, that is. Depending on what kind of difficulty you are facing, you can indicate this to the user in different ways, and there seems to be a divide between programmers as to whether one should return an error code, or do something else.

But in the python world, the most common method is to raise some sort of exception, and let the user handle the problem. In the following snippet, we will see a couple of ways of going about exceptions. We implement a single function that raises an exception, no matter what. When developing user-friendly code, that is as vicious as you can get, I guess.

First, the code listing:

https://github.com/v923z/micropython-usermod/tree/master/snippets/sillyerrors/sillyerrors.c

```
#include "py/obj.h"
#include "py/builtin.h"
#include "py/runtime.h"
#include <stdlib.h>

STATIC mp_obj_t mean_function(mp_obj_t error_code) {
    int e = mp_obj_get_int(error_code);
    if(e == 0) {
        mp_raise_msg(&mp_type_ZeroDivisionError, "thou shall not try to divide by 0␣
↪on a microcontroller!");
    } else if(e == 1) {
        mp_raise_msg(&mp_type_IndexError, "dude, that was a silly mistake!");
    } else if(e == 2) {
        mp_raise_TypeError("look, chap, you can't be serious!");
    } else if(e == 3) {
        mp_raise_OSError(e);
    } else if(e == 4) {
        char *buffer;
        buffer = malloc(100);
        sprintf(buffer, "you are really out of luck today: error code %d", e);
        mp_raise_NotImplementedError(buffer);
```

(continues on next page)

(continued from previous page)

```
    } else {
        mp_raise_ValueError("sorry, you've exhausted all your options");
    }
    return mp_const_false;
}

STATIC MP_DEFINE_CONST_FUN_OBJ_1(mean_function_obj, mean_function);

STATIC const mp_rom_map_elem_t sillyerrors_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_sillyerrors) },
    { MP_ROM_QSTR(MP_QSTR_mean), MP_ROM_PTR(&mean_function_obj) },
};
STATIC MP_DEFINE_CONST_DICT(sillyerrors_module_globals, sillyerrors_module_globals_
↪table);

const mp_obj_module_t sillyerrors_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&sillyerrors_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_sillyerrors, sillyerrors_user_cmodule, MODULE_SILLYERRORS_
↪ENABLED);
```

Now, not all exceptions are created equal. Some are more exceptional than the others: `ValueError`, `TypeError`, `OSError`, and `NotImplementedError` can be raised with the syntax

```
mp_raise_ValueError("wrong value");
```

which will, in addition to raising the exception at the C level (i.e., interrupting the execution of the code), also return a pretty traceback:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: wrong value
```

with the error message that we supplied to the `mp_raise_ValueError` function. If you want to have a traceback message that is not a compile-time constant, you could deal with the problem as in case 4 in the function `mean_function`. Such a message might be useful, if the nature of the exception is somehow related to a quantity that is not known at compile time, e.g., if you have a function that should not ever run, if the up-time is shorter than some predefined value. Of course, one can just say that the "microcontroller hasn't run long enough yet", and this is a pretty good constant string, but perhaps we can give the user a bit more information, if we can also indicate, how much time is still missing.

Other exceptions can be raised as in the `e == 1` case, with the `mp_raise_msg(&mp_type_IndexError, "dude, that was a silly mistake!")` function. Here one also has to specify the type of the exception, which is always of the form `mp_type_`. A complete list can be found in `obj.h`.

Incidentally, `mp_raise_ValueError`, `mp_raise_TypeError`, and `mp_raise_NotImplementedError` are nothing but a wrapper for `mp_raise_msg`, which in turn is a wrapper for `nlr_raise` of `nlr.c/nlr.h`. The `OSError` is somewhat curious in this respect, because it is raised directly through `nlr_raise`, and its argument is not a string, but an integer error code. All these wrappers are defined in `runtime.c`, by the way.

In our ultimate mean function, we raised a lot of exceptions by now, but we still have to return some value, because the function signature stipulates that, and the compiler would be unsatisfied otherwise, even though code execution will actually never reach the return statement. Since we are in denial mode anyway, I cast my vote for a return value of `mp_const_false`. `mp_const_none` was the other candidate, but ended up as the runner-up.

I think, it is high time to compile our code.

https://github.com/v923z/micropython-usermod/tree/master/snippets/sillyerrors/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/sillyerrors.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import sillyerrors
print(sillyerrors.mean(0))
```

```
Traceback (most recent call last):
  File "/dev/shm/micropython.py", line 3, in <module>
ZeroDivisionError: thou shall not try to divide by 0 on a microcontroller!
```

```
%%micropython

import sillyerrors
print(sillyerrors.mean(1))
```

```
Traceback (most recent call last):
  File "/dev/shm/micropython.py", line 3, in <module>
IndexError: dude, that was a silly mistake!
```

```
%%micropython

import sillyerrors
print(sillyerrors.mean(2))
```

```
Traceback (most recent call last):
  File "/dev/shm/micropython.py", line 3, in <module>
TypeError: look, chap, you can't be serious!
```

```
%%micropython

import sillyerrors
print(sillyerrors.mean(3))
```

```
Traceback (most recent call last):
  File "/dev/shm/micropython.py", line 3, in <module>
OSError: 3
```

```
%%micropython

import sillyerrors
print(sillyerrors.mean(4))
```

```
Traceback (most recent call last):
  File "/dev/shm/micropython.py", line 3, in <module>
NotImplementedError: you are really out of luck today: error code 4
```

One can't but wonder, why we had to invoke our `mean` function in four separate statements, and why we couldn't execute everything in a nice nifty package like

```
%%micropython

import sillyerrors
print(sillyerrors.mean(0))
print(sillyerrors.mean(1))
print(sillyerrors.mean(2))
print(sillyerrors.mean(3))
print(sillyerrors.mean(4))
```

```
Traceback (most recent call last):
  File "/dev/shm/micropython.py", line 3, in <module>
ZeroDivisionError: you shall not try to divide by 0 on a microcontroller!
```

Well, we could have, but since we specifically raised an exception in the first statement, our code would never have gotten beyond

```
sillyerror.mean(0)
```

After all, this is what exceptions do: they interrupt the execution of the code.

CHAPTER 6

Parsing arguments

In practically all cases, you will have to inspect the arguments of your function. Even if you can resort to functions in the micropython implementation, that simply means that the burden of inspection was taken off your shoulders, but not that the inspection does not happen at all. In this section, we are going to see what we can do with both position, and keyword arguments, and how we can retrieve their values.

6.1 Positional arguments

6.1.1 Known number of arguments

A known number of positional arguments are pretty much a done deal: we have seen how to get the C values of such arguments: in our very first module, we called `mp_obj_get_int()`, because we wanted to sum two integers. Should we like to work with float, we could call `mp_obj_get_float()`. (This function will properly work, if the value is an integer, by the way.)

If we have a more complicated construct, like a tuple or a list, we can turn the argument into a pointer with

```
mp_obj_t some_function(mp_obj_t object_in) {
    mp_obj_tuple_t *object = MP_OBJ_TO_PTR(object_in);
    ...
}
```

and continue with `*object`. We can then retrieve the tuple's structure members with `object->items` (the elements in the tuple), and `object->len` (the length of the tuple). This procedure works even with newly-defined object types. A complete example can be found in Section *Creating new types*:

```
typedef struct _vector_obj_t {
    mp_obj_base_t base;
    float x, y, z;
} vector_obj_t;
```

(continues on next page)

(continued from previous page)

```
mp_obj_t some_function(mp_obj_t object_in) {
    vector_obj_t *vector = MP_OBJ_TO_PTR(object_in);
    ...
}
```

6.1.2 Unknown number of arguments

Now, we pointed out that the macros generating the function objects can be of the form

```
MP_DEFINE_CONST_FUN_OBJ_VAR_BETWEEN(some_function_obj, n_argmin, n_argmax, some_
↪function);
```

In such a case, we surely can't just enumerate the arguments of the function without any checks, especially, that we don't even know how far we have to go, and the behaviour of the function can depend on the number of arguments. What shall we do in such an instance?

We have to reckon that the signature of a function with a variable number of arguments looks like

```
mp_obj_t some_function(size_t n_args, const mp_obj_t *args) {
    if (n_args == 2) {
        ...
    }
    ...
}
```

and the first argument of the C function will store the number of positional arguments of the python function. Once `n_args` is known, we are set. It is important to note that the work is done by the `MP_DEFINE_CONST_FUN_OBJ_VAR_BETWEEN` macro, we do not have to set up the C function in any particular way.

Here is a small example that will drive this point home.

https://github.com/v923z/micropython-usermod/tree/master/snippets/vararg/vararg.c

```
#include "py/obj.h"
#include "py/runtime.h"

STATIC mp_obj_t vararg_function(size_t n_args, const mp_obj_t *args) {
    if(n_args == 0) {
        printf("no arguments supplied\n");
    } else if(n_args == 1) {
        printf("this is a %lu\n", mp_obj_get_int(args[0]));
    } else if(n_args == 2) {
        printf("hm, we will sum them: %lu\n", mp_obj_get_int(args[0]) + mp_obj_get_
↪int(args[1]));
    } else if(n_args == 3) {
        printf("Look at that! A triplet: %lu, %lu, %lu\n", mp_obj_get_int(args[0]),␣
↪mp_obj_get_int(args[1]), mp_obj_get_int(args[2]));
    }
    return mp_const_none;
}

STATIC MP_DEFINE_CONST_FUN_OBJ_VAR_BETWEEN(vararg_function_obj, 0, 3, vararg_
↪function);
```

(continues on next page)

(continued from previous page)

```
STATIC const mp_rom_map_elem_t vararg_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_vararg) },
    { MP_ROM_QSTR(MP_QSTR_vararg), MP_ROM_PTR(&vararg_function_obj) },
};
STATIC MP_DEFINE_CONST_DICT(vararg_module_globals, vararg_module_globals_table);

const mp_obj_module_t vararg_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&vararg_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_vararg, vararg_user_cmodule, MODULE_VARARG_ENABLED);
```

https://github.com/v923z/micropython-usermod/tree/master/snippets/vararg/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/vararg.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import vararg

vararg.vararg()
vararg.vararg(1)
vararg.vararg(10, 20)
vararg.vararg(1, 22, 333)
```

```
no arguments supplied
this is a 1
hm, we will sum them: 30
Look at that! A triplet: 1, 22, 333
```

6.2 Working with strings

We have discussed numerical values in micropython at length. We know how we convert an `mp_obj_t` object to a native C type, and we also know, how we can turn an integer or float into an `mp_obj_t`, and return it at the end of the function. The key components were the `mp_obj_get_int()`, `mp_obj_new_int()`, and `mp_obj_get_float()`, and `mp_obj_new_float()` functions. Later we will see, what we can do with various iterables, like lists and tuples, but before that, I would like to explain, how one handles strings. (Strings are also iterables in python, by the way, however, they also have a native C equivalent.)

At the beginning, we said that in micropython, almost everything is an `mp_obj_t` object. Strings are no exception: however, the `mp_obj_t` that denotes the string does not store its value, but a pointer to the memory location, where the characters are stored. The reason is rather trivial: the `mp_obj_t` has a size of 8 bytes, hence, the object can't possibly store a string that is longer than 7 bytes. (The same applies to more complicated objects, e.g., lists, or tuples.)

Now, the procedure of working with the string would kick out with retrieving the pointer, and then we could increment its value till we encounter the `\0` character, which indicates that the string has ended. Fortunately, micropython has a handy macro for retrieving the string's value and its length, so we don't have to concern ourselves with the really low-level stuff. For the string utilities, we should include `py/objstr.h` (for the micropython things), and `string.h` (for `strcpy`). `py/objstr.c` contains a number of tools for string manipulation. Before you try to implement your own functions, it might be worthwhile to check that out. You might find something useful.

Our next module is going to take a single string as an argument, print out its length (you already know, how to return the length, don't you?), and return the contents backwards. All this in 33 lines.

https://github.com/v923z/micropython-usermod/tree/master/snippets/stringarg/stringarg.c

```
#include <string.h>
#include "py/obj.h"
#include "py/runtime.h"
#include "py/objstr.h"

#define byteswap(a,b) char tmp = a; a = b; b = tmp;

STATIC mp_obj_t stringarg_function(const mp_obj_t o_in) {
    mp_check_self(mp_obj_is_str_or_bytes(o_in));
    GET_STR_DATA_LEN(o_in, str, str_len);
    printf("string length: %lu\n", str_len);
    char out_str[str_len];
    strcpy(out_str, (char *)str);
    for(size_t i=0; i < (str_len-1)/2; i++) {
        byteswap(out_str[i], out_str[str_len-i-1]);
    }
    return mp_obj_new_str(out_str, str_len);
}

STATIC MP_DEFINE_CONST_FUN_OBJ_1(stringarg_function_obj, stringarg_function);

STATIC const mp_rom_map_elem_t stringarg_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_stringarg) },
    { MP_ROM_QSTR(MP_QSTR_stringarg), MP_ROM_PTR(&stringarg_function_obj) },
};
STATIC MP_DEFINE_CONST_DICT(stringarg_module_globals, stringarg_module_globals_table);

const mp_obj_module_t stringarg_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&stringarg_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_stringarg, stringarg_user_cmodule, MODULE_STRINGARG_
↪ENABLED);
```

The macro defined in `objstr.h` takes three arguments, out of which only the first one is actually defined. The other two are defined in the macro itself. So, in the line

```
GET_STR_DATA_LEN(o_in, str, str_len);
```

only `o_in` is known at the moment the macro is called, `str`, which will be a pointer to type character, and `str_len`, which is of type `size_t`, and holds the length of the string, are created by `GET_STR_DATA_LEN` itself. This is, why we can later stick `str_len`, and `str` into print statements, though, we never declared these variables.

After `GET_STR_DATA_LEN` has been called, we are in C land. First, we print out the length, then reverse the string. But why can't we do the string inversion on the original string, and why do we have to declare a new variable, `out_str`? The reason for that is that the `GET_STR_DATA_LEN` macro declares a `const` string, which we can't

change anymore, so we have to copy the content (`strcpy` from `string.h`), and swap the bytes in `out_str`. When doing so, we should keep in mind that the very last byte in the string is the termination character, hence, we exchange the `i`th position with the `str_len-i-1`th position. If you fail to notice the `-1`, you'll end up with an empty string: even though the byte swapping would run without complaints, the very first byte would be equal to `\0`.

At the very end, we return from our function with a call to `mp_obj_new_str`, which creates a new `mp_obj_t` object that points to the content of the string. And we are done! All there is left to do is compilation. Let's take care of that!

https://github.com/v923z/micropython-usermod/tree/master/snippets/stringarg/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/stringarg.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import stringarg

print(stringarg.stringarg('...krow ta eludom gragnirts eht'))
```

```
string length: 31
the stringarg module at work...
```

6.3 Keyword arguments

One of the most useful features of python is that functions can accept positional as well as keyword arguments, thereby providing a very flexible and instructive function interface. (Instructive, insofar as the intent of a variable is very explicit, even at the user level.) In this subsection, we will learn how the processing of keyword arguments is done. Our new module will be the sexed-up version of our very first one, where we added two integers. We will do the same here, except that the second argument will be a keyword, and will assume a default value of 0.

Before jumping into the implementation, we should contemplate the task for a second. It does not matter, whether we have positional or keyword arguments, at one point, the interpreter has to turn all arguments into a deterministic sequence of objects. We stipulate this sequence in the constant variable called `allowed_args[]`. This is an array of type `mp_arg_t`, which is nothing but a structure with two `uint16` values, and a union named `mp_arg_val_t`. This union holds the default value and the type of the variable that we want to pass. The `mp_arg_t` structure, defined in `runtime.h`, looks like this:

```
typedef struct _mp_arg_t {
    uint16_t qst;
    uint16_t flags;
    mp_arg_val_t defval;
} mp_arg_t;
```

The last member, `mp_arg_val_t` is

```
typedef union _mp_arg_val_t {
    bool u_bool;
    mp_int_t u_int;
    mp_obj_t u_obj;
    mp_rom_obj_t u_rom_obj;
} mp_arg_val_t;
```

Keyword arguments come in three flavours: `MP_ARG_BOOL`', `MP_ARG_INT`, and `MP_ARG_OBJ`.

6.3.1 Keyword arguments with numerical values

And now the implementation:

https://github.com/v923z/micropython-usermod/tree/master/snippets/keywordfunction/keywordfunction.c

```
#include <stdio.h>
#include "py/obj.h"
#include "py/runtime.h"
#include "py/builtin.h"

STATIC mp_obj_t keywordfunction_add_ints(size_t n_args, const mp_obj_t *pos_args, mp_
↪map_t *kw_args) {
    static const mp_arg_t allowed_args[] = {
        { MP_QSTR_a, MP_ARG_REQUIRED | MP_ARG_INT, {.u_int = 0} },
        { MP_QSTR_b, MP_ARG_KW_ONLY | MP_ARG_INT, {.u_int = 0} },
    };

    mp_arg_val_t args[MP_ARRAY_SIZE(allowed_args)];
    mp_arg_parse_all(n_args, pos_args, kw_args, MP_ARRAY_SIZE(allowed_args), allowed_
↪args, args);
    int16_t a = args[0].u_int;
    int16_t b = args[1].u_int;
    printf("a = %d, b = %d\n", a, b);
    return mp_obj_new_int(a + b);
}

STATIC MP_DEFINE_CONST_FUN_OBJ_KW(keywordfunction_add_ints_obj, 1, keywordfunction_
↪add_ints);

STATIC const mp_rom_map_elem_t keywordfunction_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_keywordfunction) },
    { MP_ROM_QSTR(MP_QSTR_add_ints), (mp_obj_t)&keywordfunction_add_ints_obj },
};

STATIC MP_DEFINE_CONST_DICT(keywordfunction_module_globals, keywordfunction_module_
↪globals_table);

const mp_obj_module_t keywordfunction_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&keywordfunction_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_keywordfunction, keywordfunction_user_cmodule, MODULE_
↪KEYWORDFUNCTION_ENABLED);
```

One side effect of a function with keyword arguments is that we do not have to care about the arguments in the C implementation: the argument list is always the same, and it is passed in by the interpreter: the number of arguments

of the python function, an array with the positional arguments, and a map for the keyword arguments.

After parsing the arguments with `mp_arg_parse_all`, whatever was at the zeroth position of `allowed_args[]` will be called `args[0]`, the object at the first position of `allowed_args[]` will be turned into `args[1]`, and so on.

This is, where we also define, what the name of the keyword argument is going to be: whatever comes after `MP_QSTR_`. But hey, presto! The name should be an integer with 16 bits, shouldn't it? After all, this is the first member of `mp_arg_t`. So what the hell is going on here? Well, for the efficient use of RAM, all MP_QSTRs are turned into `unint16_t` internally. This applies not only to the names in functions with keyword arguments, but also for module and function names, in the `_module_globals_table[]`.

The second member of the `mp_arg_t` structure is the flags that determine, e.g., whether the argument is required, if it is of integer or `mp_obj_t` type, and whether it is a positional or a keyword argument. These flags can be combined by ORing them, as we have done in the example above.

The last member in `mp_arg_t` is the default value. Since this is a member variable, when we make use of it, we have to extract the value by adding `.u_int` to the argument.

When turning our function into a function object, we have to call a special macro, `MP_DEFINE_CONST_FUN_OBJ_KW`, defined in `obj.h`, which is somewhat similar to `MP_DEFINE_CONST_FUN_OBJ_VAR`: in addition to the function object and the function, one also has to specify the minimum number of arguments in the python function.

Other examples on passing keyword arguments can be found in some of the hardware implementation files, e.g., `ports/stm32/pyb_i2c.c`, or `ports/stm32/pyb_spi.c`.

Now, let us see, whether we can add two numbers here.

https://github.com/v923z/micropython-usermod/tree/master/snippets/keywordfunction/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/keywordfunction.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import keywordfunction
print(keywordfunction.add_ints(-3, b=4))
print(keywordfunction.add_ints(3))
```

```
a = -3, b = 4
1
a = 3, b = 0
3
```

As advertised, both function calls do what they were supposed to do: in the first case, `b` assumes the value of 4, while in the second case, it takes on 0, even though we didn't supply anything to the function.

6.3.2 Arbitrary keyword arguments

We have seen how integer values can be extracted from keyword arguments, but unfortunately, that method is going to get you only that far. What if we want to pass something more complicated, in particular a string, or a tuple, or some other non-trivial python type?

A simple solution could be to implement the C function without keywords at all, and do the parsing in python. After all, it is highly unlikely that parsing would be expensive in comparison to the body of the function. But perhaps, you have your reasons for not going down that rabbit hole.

For such cases, we can still resort to objects of type `.u_rom_obj`. In order to experiment with the possibilities, in the next module, we define a function that simply returns the values passed to it. The input arguments are going to be a single positional argument, and four keyword arguments with type `int`, `string`, `tuple`, and `float`.

https://github.com/v923z/micropython-usermod/tree/master/snippets/arbitrarykeyword/arbitrarykeyword.c

```
#include <stdio.h>
#include "py/obj.h"
#include "py/objlist.h"
#include "py/runtime.h"
#include "py/builtin.h"

// This is lifted from objfloat.c, because mp_obj_float_t is not exposed there (there␣
↪is no header file)
typedef struct _mp_obj_float_t {
    mp_obj_base_t base;
    mp_float_t value;
} mp_obj_float_t;

const mp_obj_float_t my_float = {{&mp_type_float}, 0.987};

const mp_rom_obj_tuple_t my_tuple = {
    {&mp_type_tuple},
    3,
    {
        MP_ROM_INT(0),
        MP_ROM_QSTR(MP_QSTR_float),
        MP_ROM_PTR(&my_float),
    },
};

STATIC mp_obj_t arbitrarykeyword_print(size_t n_args, const mp_obj_t *pos_args, mp_
↪map_t *kw_args) {
    static const mp_arg_t allowed_args[] = {
        { MP_QSTR_a, MP_ARG_INT, {.u_int = 0} },
        { MP_QSTR_b, MP_ARG_KW_ONLY | MP_ARG_INT, {.u_int = 1} },
        { MP_QSTR_c, MP_ARG_KW_ONLY | MP_ARG_OBJ, {.u_rom_obj = MP_ROM_QSTR(MP_QSTR_
↪float)} },
        { MP_QSTR_d, MP_ARG_KW_ONLY | MP_ARG_OBJ, {.u_rom_obj = MP_ROM_PTR(&my_float)}
↪ },
        { MP_QSTR_e, MP_ARG_KW_ONLY | MP_ARG_OBJ, {.u_rom_obj = MP_ROM_PTR(&my_tuple)}
↪ },
    };

    mp_arg_val_t args[MP_ARRAY_SIZE(allowed_args)];
    mp_arg_parse_all(1, pos_args, kw_args, MP_ARRAY_SIZE(allowed_args), allowed_args,␣
↪args);
    mp_obj_t tuple[5];
```

(continues on next page)

(continued from previous page)

```
    tuple[0] = mp_obj_new_int(args[0].u_int); // a
    tuple[1] = mp_obj_new_int(args[1].u_int); // b
    tuple[2] = args[2].u_obj; // c
    tuple[3] = args[3].u_obj; // d
    tuple[4] = args[4].u_obj; // e
    return mp_obj_new_tuple(5, tuple);
}

STATIC MP_DEFINE_CONST_FUN_OBJ_KW(arbitrarykeyword_print_obj, 1, arbitrarykeyword_
↪print);

STATIC const mp_rom_map_elem_t arbitrarykeyword_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_arbitrarykeyword) },
    { MP_ROM_QSTR(MP_QSTR_print), (mp_obj_t)&arbitrarykeyword_print_obj },
};

STATIC MP_DEFINE_CONST_DICT(arbitrarykeyword_module_globals, arbitrarykeyword_module_
↪globals_table);

const mp_obj_module_t arbitrarykeyword_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&arbitrarykeyword_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_arbitrarykeyword, arbitrarykeyword_user_cmodule, MODULE_
↪ARBITRARYKEYWORD_ENABLED);
```

Before compiling the code, let us think a bit about what is going on here. The first argument, `a`, is straightforward: that is a positional argument, and we deal with that as we did in the last example. The same applies to the second argument, `b`, which is our first keyword argument with an integer default value.

Matters become more interesting with the third argument, `c`: that is supposed to be a string, whose default value is "float". We generate the respective C representation by prepending the `MP_QSTR_`. At this point, we have a string, but we still can't assign it as a default value. We do that by first applying the `MP_ROM_QSTR` macro, and assigning the results to the `.u_rom_obj` member of the `mp_arg_t` structure. You most certainly will want to inspect the value at one point. We have already discussed the drill in *Working with strings*.

The fourth argument, `d`, is meant to be a float. Since there is no equivalent of a float in the `mp_arg_t` structure, we have to turn our number into an `MP_ROM_PTR`, so we have to retrieve the address of the float object. To this end, we define the number in the line

```
const mp_obj_float_t my_float = {{&mp_type_float}, 0.987};
```

Note that since `mp_obj_float_t` is not exposed in `objfloat.c`, where it is defined, we had to copy the type declaration. This is certainly not very elegant, but desperate times call for desperate measures. In addition, we also have to declare `my_float` as a constant. The reason for this is that we have to assure the compiler that this value is not going to change in the future, so that it can be saved into the read-only memory.

The last argument, `e`, is a tuple, which has a special type for such cases, namely, the `mp_rom_obj_tuple_t`, so we define `my_tuple` as an `mp_rom_obj_tuple_t` object, with a base type of `mp_type_tuple`, and three elements, an integer, a string, and a float. The elements go into the tuple as if they were assigned to the `.u_rom_obj` members directly, hence the macros `MP_ROM_INT`, `MP_ROM_QSTR`, and `MP_ROM_PTR`.

When we return the default values at the end of our function, we declare an array of type `mb_obj_t`, and of length 5, assign the elements, and turn the array into a tuple with `mp_obj_new_tuple`.

One final comment to this section: I referred to our function as returning the values of the arguments, yet, I called it `print`. Had I called the function `return`, it wouldn't have worked for the simple reason, that `return` is a keyword

of the language itself. As a friendly advice, do not try to override that!

Having thoroughly discussed the code, we should compile it, and see what happens.

https://github.com/v923z/micropython-usermod/tree/master/snippets/arbitrarykeyword/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/arbitrarykeyword.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import arbitrarykeyword
print(arbitrarykeyword.print(1, b=123))
print(arbitrarykeyword.print(-35, b=555, c='foo', d='bar', e=[1, 2, 3]))
```

```
(1, 123, 'float', 0.9869999999999999, (0, 'float', 0.9869999999999999))
(-35, 555, 'foo', 'bar', [1, 2, 3])
```

CHAPTER 7

Working with classes

Of course, python would not be python without classes. A module can also include the implementation of classes. The procedure is similar to what we have already seen in the context of standard functions, except that we have to define a structure that holds at least a string with the name of the class, a pointer to the initialisation and printout functions, and a local dictionary. A typical class structure would look like

```
STATIC const mp_rom_map_elem_t simpleclass_locals_dict_table[] = {
    { MP_ROM_QSTR(MP_QSTR_method1), MP_ROM_PTR(&simpleclass_method1_obj) },
    { MP_ROM_QSTR(MP_QSTR_method2), MP_ROM_PTR(&simpleclass_method2_obj) },
    ...
}

const mp_obj_type_t simpleclass_type = {
    { &mp_type_type },
    .name = MP_QSTR_simpleclass,
    .print = simpleclass_print,
    .make_new = simpleclass_make_new,
    .locals_dict = (mp_obj_dict_t*)&simpleclass_locals_dict,
};
```

The locals dictionary, `.locals_dict`, contains all user-facing methods and constants of the class, while the `simpleclass_type` structure's `name` member is what our class is going to be called. `.print` is roughly the equivalent of `__str__`, and `.make_new` is the C name for `__init__`.

In order to see how this all works, we are going to implement a very simple class, which holds two integer variables, and has a method that returns the sum of these two variables. In python, a possible realisation could look like this:

```
class myclass:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def mysum(self):
        return self.a + self.b
```

(continues on next page)

(continued from previous page)

```
A = myclass(1, 2)
A.mysum()
```

```
3
```

In addition to the class implementation above and in order to show how class methods and regular functions can live in the same module, we will also have a function, which is not bound to the class itself, and which adds the two components in the class, i.e., that is similar to

```
def add(class_instance):
    return class_instance.a + class_instance.b

add(A)
```

```
3
```

(Note that retrieving values from the class in this way is not exactly elegant, nor is it pythonic. We would usually implement a getter method for that.)

https://github.com/v923z/micropython-usermod/tree/master/snippets/simpleclass/simpleclass.c

```
#include <stdio.h>
#include "py/runtime.h"
#include "py/obj.h"

typedef struct _simpleclass_myclass_obj_t {
    mp_obj_base_t base;
    int16_t a;
    int16_t b;
} simpleclass_myclass_obj_t;

const mp_obj_type_t simpleclass_myclass_type;

STATIC void myclass_print(const mp_print_t *print, mp_obj_t self_in, mp_print_kind_t␣
↪kind) {
    (void)kind;
    simpleclass_myclass_obj_t *self = MP_OBJ_TO_PTR(self_in);
    printf("myclass(%d, %d)", self->a, self->b);
}

STATIC mp_obj_t myclass_make_new(const mp_obj_type_t *type, size_t n_args, size_t n_
↪kw, const mp_obj_t *args) {
    mp_arg_check_num(n_args, n_kw, 2, 2, true);
    simpleclass_myclass_obj_t *self = m_new_obj(simpleclass_myclass_obj_t);
    self->base.type = &simpleclass_myclass_type;
    self->a = mp_obj_get_int(args[0]);
    self->b = mp_obj_get_int(args[1]);
    return MP_OBJ_FROM_PTR(self);
}

// Class methods
STATIC mp_obj_t myclass_sum(mp_obj_t self_in) {
    simpleclass_myclass_obj_t *self = MP_OBJ_TO_PTR(self_in);
    return mp_obj_new_int(self->a + self->b);
```

(continues on next page)

(continued from previous page)

```
}

MP_DEFINE_CONST_FUN_OBJ_1(myclass_sum_obj, myclass_sum);

STATIC const mp_rom_map_elem_t myclass_locals_dict_table[] = {
    { MP_ROM_QSTR(MP_QSTR_mysum), MP_ROM_PTR(&myclass_sum_obj) },
};

STATIC MP_DEFINE_CONST_DICT(myclass_locals_dict, myclass_locals_dict_table);

const mp_obj_type_t simpleclass_myclass_type = {
    { &mp_type_type },
    .name = MP_QSTR_simpleclass,
    .print = myclass_print,
    .make_new = myclass_make_new,
    .locals_dict = (mp_obj_dict_t*)&myclass_locals_dict,
};

// Module functions
STATIC mp_obj_t simpleclass_add(const mp_obj_t o_in) {
    simpleclass_myclass_obj_t *class_instance = MP_OBJ_TO_PTR(o_in);
    return mp_obj_new_int(class_instance->a + class_instance->b);
}

MP_DEFINE_CONST_FUN_OBJ_1(simpleclass_add_obj, simpleclass_add);

STATIC const mp_map_elem_t simpleclass_globals_table[] = {
    { MP_OBJ_NEW_QSTR(MP_QSTR___name__), MP_OBJ_NEW_QSTR(MP_QSTR_simpleclass) },
    { MP_OBJ_NEW_QSTR(MP_QSTR_myclass), (mp_obj_t)&simpleclass_myclass_type },
    { MP_OBJ_NEW_QSTR(MP_QSTR_add), (mp_obj_t)&simpleclass_add_obj },
};

STATIC MP_DEFINE_CONST_DICT (
    mp_module_simpleclass_globals,
    simpleclass_globals_table
);

const mp_obj_module_t simpleclass_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&mp_module_simpleclass_globals,
};

MP_REGISTER_MODULE(MP_QSTR_simpleclass, simpleclass_user_cmodule, MODULE_SIMPLECLASS_
↪ENABLED);
```

In `my_print`, we used the C function `printf`, but better options are also available. `mpprint.c` has a number of methods for printing all kinds of python objects.

One more thing to note: the functions that are pointed to in `simpleclass_myclass_type` are not registered with the macro `MP_DEFINE_CONST_FUN_OBJ_VAR` or similar. The reason for this is that this automatically happens: `myclass_print` does not require user-supplied arguments beyond `self`, so it is known what the signature should look like. In `myclass_make_new`, we inspect the argument list, when calling

```
mp_arg_check_num(n_args, n_kw, 2, 2, true);
```

so, again, there is no need to turn our function into a function object.

https://github.com/v923z/micropython-usermod/tree/master/snippets/simpleclass/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/simpleclass.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import simpleclass
a = simpleclass.myclass(2, 3)
print(a)
print(a.mysum())
```

```
myclass(2, 3)
5
```

7.1 Special methods of classes

Python has a number of special methods, which will make a class behave as a native object. So, e.g., if a class implements the `__add__(self, other)` method, then instances of that class can be added with the + operator. Here is an example in python:

```
class Adder:

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

    def __add__(self, other):
        self.value = self.value + other.value
        return self

a = Adder(1)
b = Adder(2)

c = a + b
c.value
```

```
3
```

Note that, while the above example is not particularly useful, it proves the point: upon calling the + operator, the values of `a`, and `b` are added. If we had left out the implementation of the `__add__` method, the python interpreter would not have a clue as to what to do with the objects. You can see for yourself, how sloppiness makes itself manifest:

```
class Adder:

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

```

(continues on next page)

(continued from previous page)

```
a = Adder(1)
b = Adder(2)

c = a + b
c.value
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-77-635006a6f7bc> in <module>
      7 b = Adder(2)
      8
----> 9 c = a + b
     10 c.value


TypeError: unsupported operand type(s) for +: 'Adder' and 'Adder'
```

Indeed, we do not support the + operator.

Now, the problem is that in the C implementation, these special methods have to be treated in a special way. The naive approach would be to add the pointer to the function to the locals dictionary as

```
STATIC const mp_rom_map_elem_t simpleclass_locals_dict_table[] = {
    { MP_ROM_QSTR(MP_QSTR___add__), MP_ROM_PTR(&simpleclass_add_obj) },
};
```

but that would not work. Well, this is not entirely true: the + operator would not work, but one could still call the method explicitly as

```
a = Adder(1)
b = Adder(2)

a.__add__(b)
```

Before we actually add the + operator to our class, we should note that there are two kinds of special methods, namely the unary and the binary operators.

In the first group are those, whose sole argument is the class instance itself. Two frequently used cases are the length operator, `len`, and `bool`. So, e.g., if your class implements the `__len__(self)` method, and the method returns an integer, then you can call the `len` function in the console

```
len(myclass)
```

In the second category of operators are those, which require a left, as well as a right hand side: the operand on the left hand side is the class instance itself, while the right hand side can, in principle, be another instance of the same class, or some other type. An example for this was the `__add__` method in our `Adder` class. To prove that the right hand side needn't be of the same type, think of the *multiplication* of lists:

```
[1, 2, 3]*5
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

is perfectly valid, and has a well-defined meaning. It is the responsibility of the C implementation to inspect the right hand side, and decide how to interpret the operation. The complete list of unary, as well as binary operators can be

found in `runtime.h`.

The module below implements five special methods altogether. Two unary, namely, `bool`, and `len`, and three binary operators, `==`, `+`, and `*`. Since the addition and multiplication will return a new instance of `specialclass_myclass`, we define a new function, `create_new_class`, that, well, creates a new instance of `specialclass_myclass`, and initialises the members with the two input arguments. This function will also be called in the class initialisation function, `myclass_make_new`, immediately after the argument checking.

When implementing the operators, we have to keep a couple of things in mind. First, the `specialclass_myclass_type` has to be extended with the two methods, `.unary_op`, and `.binary_op`, where `.unary_op` is equal to the function that handles the unary operation (`specialclass_unary_op` in the example below), and `.binary_op` is equal to the function that deals with binary operations (`specialclass_binary_op` below). These two functions have the signatures

```
STATIC mp_obj_t specialclass_unary_op(mp_unary_op_t op, mp_obj_t self_in)
```

and

```
STATIC mp_obj_t specialclass_binary_op(mp_binary_op_t op, mp_obj_t lhs, mp_obj_t rhs)
```

respectively, and we have to inspect the value of `op` in the implementation. This is done in the two `switch` statements.

Second, if `.unary_op`, or `.binary_op` are defined for the class, then the handler function must have an implementation of all possible operators. This doesn't necessarily mean that you have to have all cases in the `switch`, but if you haven't, then there must be a `default` case with a reasonable return value, e.g., `MP_OBJ_NULL`, or `mp_const_none`, so as to indicate that that particular method is not available.

https://github.com/v923z/micropython-usermod/tree/master/snippets/specialclass/specialclass.c

```
#include <stdio.h>
#include "py/runtime.h"
#include "py/obj.h"
#include "py/binary.h"

typedef struct _specialclass_myclass_obj_t {
    mp_obj_base_t base;
    int16_t a;
    int16_t b;
} specialclass_myclass_obj_t;

const mp_obj_type_t specialclass_myclass_type;

STATIC void myclass_print(const mp_print_t *print, mp_obj_t self_in, mp_print_kind_t␣
↪kind) {
    (void)kind;
    specialclass_myclass_obj_t *self = MP_OBJ_TO_PTR(self_in);
    printf("myclass(%d, %d)", self->a, self->b);
}

mp_obj_t create_new_myclass(uint16_t a, uint16_t b) {
    specialclass_myclass_obj_t *out = m_new_obj(specialclass_myclass_obj_t);
    out->base.type = &specialclass_myclass_type;
    out->a = a;
    out->b = b;
    return MP_OBJ_FROM_PTR(out);
}

STATIC mp_obj_t myclass_make_new(const mp_obj_type_t *type, size_t n_args, size_t n_
↪kw, const mp_obj_t *args) {
```

(continues on next page)

(continued from previous page)

```
    mp_arg_check_num(n_args, n_kw, 2, 2, true);
    return create_new_myclass(mp_obj_get_int(args[0]), mp_obj_get_int(args[1]));
}

STATIC const mp_rom_map_elem_t myclass_locals_dict_table[] = {
};

STATIC MP_DEFINE_CONST_DICT(myclass_locals_dict, myclass_locals_dict_table);

STATIC mp_obj_t specialclass_unary_op(mp_unary_op_t op, mp_obj_t self_in) {
    specialclass_myclass_obj_t *self = MP_OBJ_TO_PTR(self_in);
    switch (op) {
        case MP_UNARY_OP_BOOL: return mp_obj_new_bool((self->a > 0) && (self->b > 0));
        case MP_UNARY_OP_LEN: return mp_obj_new_int(2);
        default: return MP_OBJ_NULL; // operator not supported
    }
}

STATIC mp_obj_t specialclass_binary_op(mp_binary_op_t op, mp_obj_t lhs, mp_obj_t rhs)
↪{
    specialclass_myclass_obj_t *left_hand_side = MP_OBJ_TO_PTR(lhs);
    specialclass_myclass_obj_t *right_hand_side = MP_OBJ_TO_PTR(rhs);
    switch (op) {
        case MP_BINARY_OP_EQUAL:
            return mp_obj_new_bool((left_hand_side->a == right_hand_side->a) && (left_
↪hand_side->b == right_hand_side->b));
        case MP_BINARY_OP_ADD:
            return create_new_myclass(left_hand_side->a + right_hand_side->a, left_
↪hand_side->b + right_hand_side->b);
        case MP_BINARY_OP_MULTIPLY:
            return create_new_myclass(left_hand_side->a * right_hand_side->a, left_
↪hand_side->b * right_hand_side->b);
        default:
            return MP_OBJ_NULL; // operator not supported
    }
}

const mp_obj_type_t specialclass_myclass_type = {
    { &mp_type_type },
    .name = MP_QSTR_specialclass,
    .print = myclass_print,
    .make_new = myclass_make_new,
    .unary_op = specialclass_unary_op,
    .binary_op = specialclass_binary_op,
    .locals_dict = (mp_obj_dict_t*)&myclass_locals_dict,
};

STATIC const mp_map_elem_t specialclass_globals_table[] = {
    { MP_OBJ_NEW_QSTR(MP_QSTR___name__), MP_OBJ_NEW_QSTR(MP_QSTR_specialclass) },
    { MP_OBJ_NEW_QSTR(MP_QSTR_myclass), (mp_obj_t)&specialclass_myclass_type },
};

STATIC MP_DEFINE_CONST_DICT (
    mp_module_specialclass_globals,
    specialclass_globals_table
);
```

(continues on next page)

(continued from previous page)

```
const mp_obj_module_t specialclass_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&mp_module_specialclass_globals,
};

MP_REGISTER_MODULE(MP_QSTR_specialclass, specialclass_user_cmodule, MODULE_
↪SPECIALCLASS_ENABLED);
```

https://github.com/v923z/micropython-usermod/tree/master/snippets/specialclass/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/specialclass.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import specialclass

a = specialclass.myclass(1, 2)
b = specialclass.myclass(10, 20)
print(a)
print(b)
print(a + b)
```

```
myclass(1, 2)
myclass(10, 20)
myclass(11, 22)
```

7.2 Defining constants

Constants can be added to the locals dictionary as any other object. So, e.g., if we wanted to define the constant MAGIC, we could do that as follows

```
#define MAGIC 42

STATIC const mp_rom_map_elem_t some_class_locals_dict_table[] = {
    { MP_ROM_QSTR(MP_QSTR_MAGIC), MP_ROM_INT(MAGIC) },
};
```

and then the constant would then be accessible in the interpreter as

```
import some_class

some_class.MAGIC
```

CHAPTER 8

Creating new types

Sometimes you might need something beyond the standard python data types, and you have to define your own. At first, the task seems daunting, but types are really nothing but a C structure with a couple of special fields. The steps required are very similar to those for classes. Take the following type definition, which could be regarded as the Cartesian components of a vector in three-dimensional space:

```
typedef struct _vector_obj_t {
    mp_obj_base_t base;
    float x, y, z;
} vector_obj_t;
```

Now, in order to see, how we can work with this structure, we are going to define a new type that simply stores the three values. The module will also have a method called `length`, returning the absolute value of the vector. Also note that here we check the type of the argument, and bail out, if it is not a vector. The beauty of all this is that once the type is defined, the available micropython methods just work. Can you still recall the

```
MP_OBJ_IS_TYPE(myobject, &my_type)
```

macro in Section *Type checking?* I thought so.

We have our vector structure at the C level. It has four members: an `mp_obj_base_t`, and three floats called `x`, `y`, and `z`. But this is still not usable in the python interpreter. We have to somehow tell the interpreter, what it is supposed to do with this new type, and how a variable of this type is to be presented to the user. This is, where the structure

```
const mp_obj_type_t vector_type = {
    { &mp_type_type },
    .name = MP_QSTR_vector,
    .print = vector_print,
    .make_new = vector_make_new,
};
```

takes centre stage. Does this look familiar? This structure contains the new type's name (a string, `vector`), how it presents itself to users (a function, `vector_print`), and how a new instance is to be created (a function, `vector_make_new`). These latter two we have to implement ourselves.

In `vector_print` we have three arguments, namely `const mp_print_t *print`, which is a helper that we don't call, `mp_obj_t self_in` which is a reference to the vector itself, and `mp_print_kind_t kind`, which we can graciously ignore, because we are not going to use it anyway.

Having seen the bits and pieces, we should build some new firmware.

https://github.com/v923z/micropython-usermod/tree/master/snippets/vector/vector.c

```
#include <math.h>
#include <stdio.h>
#include "py/obj.h"
#include "py/runtime.h"

const mp_obj_type_t vector_type;

typedef struct _vector_obj_t {
    mp_obj_base_t base;
    float x, y, z;
} vector_obj_t;

STATIC mp_obj_t vector_length(mp_obj_t o_in) {
    if(!mp_obj_is_type(o_in, &vector_type)) {
        mp_raise_TypeError("argument is not a vector");
    }
    vector_obj_t *vector = MP_OBJ_TO_PTR(o_in);
    return mp_obj_new_float(sqrtf(vector->x*vector->x + vector->y*vector->y + vector->
↪z*vector->z));
}

STATIC MP_DEFINE_CONST_FUN_OBJ_1(vector_length_obj, vector_length);

STATIC void vector_print(const mp_print_t *print, mp_obj_t self_in, mp_print_kind_t␣
↪kind) {
    (void)kind;
    vector_obj_t *self = MP_OBJ_TO_PTR(self_in);
    printf("vector(%f, %f, %f)\n", (double)self->x, (double)self->y, (double)self->z);
}

STATIC mp_obj_t vector_make_new(const mp_obj_type_t *type, size_t n_args, size_t n_kw,
↪ const mp_obj_t *args) {
    mp_arg_check_num(n_args, n_kw, 3, 3, true);

    vector_obj_t *vector = m_new_obj(vector_obj_t);
    vector->base.type = &vector_type;
    vector->x = mp_obj_get_float(args[0]);
    vector->y = mp_obj_get_float(args[1]);
    vector->z = mp_obj_get_float(args[2]);
    return MP_OBJ_FROM_PTR(vector);
}

const mp_obj_type_t vector_type = {
    { &mp_type_type },
    .name = MP_QSTR_vector,
    .print = vector_print,
    .make_new = vector_make_new,
};

STATIC const mp_rom_map_elem_t vector_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_vector) },
```

(continues on next page)

(continued from previous page)

```
    { MP_OBJ_NEW_QSTR(MP_QSTR_vector), (mp_obj_t)&vector_type },
    { MP_ROM_QSTR(MP_QSTR_length), MP_ROM_PTR(&vector_length_obj) },
};
STATIC MP_DEFINE_CONST_DICT(vector_module_globals, vector_module_globals_table);

const mp_obj_module_t vector_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&vector_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_vector, vector_user_cmodule, MODULE_VECTOR_ENABLED);
```

https://github.com/v923z/micropython-usermod/tree/master/snippets/vector/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/vector.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import vector

a = vector.vector(1, 20, 30)
print(a)
print(vector.length(a))
```

```
vector(1.000000, 20.000000, 30.000000)

36.06937789916993
```

Just to convince ourselves, when calculated in python proper, the length of the vector is

```
import math

print(math.sqrt(1**2 + 20**2 + 30**2))
```

```
36.069377593742864
```

Close enough.

CHAPTER 9

Dealing with iterables

Without going too deeply into specifics, in python, an iterable is basically an object that you can have in a `for` loop:

```
for item in my_iterable:
    print(item)
```

Amongst others, lists, tuples, and ranges are iterables, as are strings. The key is that these objects have a special internal method, an iterator, attached to them. This iterator is responsible for keeping track of the index during the iteration, and serving the objects in the iterable one by one to the `for` loop. When writing our own iterable, we will look under the hood, and see how this all works at the C level. For now, we are going to discuss only, how we can *consume* the content of an iterable in the C code.

9.1 Iterating over built-in types

In order to demonstrate the use of an iterator, we are going to write a function that sums the square of the values in an iterable. The python version of the function could be something like this:

```
def sumsq(some_iterable):
    return sum([item**2 for item in some_iterable])

sumsq([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

```
385
```

In C, the key is in the snippet

```
mp_obj_iter_buf_t iter_buf;
mp_obj_t item, iterable = mp_getiter(o_in, &iter_buf);
while ((item = mp_iternext(iterable)) != MP_OBJ_STOP_ITERATION) {
    // do something with the item just retrieved
}
```

This is more or less the equivalent of the `for item in some_iterable` instruction. In C, the `mp_obj_t` object `o_in` is the argument of our python function, which is turned into an iterable by passing it into the `mp_getiter` function. This function also needs a buffer object that is of type `mp_obj_iter_buf_t`. The buffer type is defined in `obj.h` as

```
typedef struct _mp_obj_iter_buf_t {
    mp_obj_base_t base;
    mp_obj_t buf[3];
} mp_obj_iter_buf_t;
```

where `.buf[2]` holds the index value, and this is how `mp_iternext` keeps track of the position in the loop.

Once `item` is retrieved, the rest of the code is trivial: you do whatever you want to do with the value, and return at the very end.

Now, what happens, if you pass a non-iterable object to the function? For a while, nothing. Everything will work till the point `item = mp_iternext(iterable)`, where the interpreter will raise a `TypeError` exception. So, on the python console, you can either enclose your function in a

```
try:
    sumsq(some_iterable)
except TypeError:
    print('something went terribly wrong`)
```

construct, or you can inspect the type of the variable at the C level. Unfortunately, there does not seem to be a type identifier for iterables in general, so you have to check, whether the argument is a list, tuple, range, etc. This can be done by calling the `MP_OBJ_IS_TYPE` macro, and see which Boolean it returns, if you pass `&mp_type_tuple`, `&mp_type_list`, `&mp_type_range` etc. to it, as we discussed in the section *Object representation*.

The complete code listing of `consumeiterable.c` follows below. If you ask me, this is a lot of code just to replace a python one-liner.

https://github.com/v923z/micropython-usermod/tree/master/snippets/consumeiterable/consumeiterable.c

```
#include "py/obj.h"
#include "py/runtime.h"

STATIC mp_obj_t consumeiterable_sumsq(mp_obj_t o_in) {
    mp_float_t _sum = 0.0, itemf;
    mp_obj_iter_buf_t iter_buf;
    mp_obj_t item, iterable = mp_getiter(o_in, &iter_buf);
    while ((item = mp_iternext(iterable)) != MP_OBJ_STOP_ITERATION) {
        itemf = mp_obj_get_float(item);
        _sum += itemf*itemf;
    }
    return mp_obj_new_float(_sum);
}

STATIC MP_DEFINE_CONST_FUN_OBJ_1(consumeiterable_sumsq_obj, consumeiterable_sumsq);

STATIC const mp_rom_map_elem_t consumeiterable_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_consumeiterable) },
    { MP_ROM_QSTR(MP_QSTR_sumsq), MP_ROM_PTR(&consumeiterable_sumsq_obj) },
};
STATIC MP_DEFINE_CONST_DICT(consumeiterable_module_globals, consumeiterable_module_
↪globals_table);

const mp_obj_module_t consumeiterable_user_cmodule = {
```

(continues on next page)

(continued from previous page)

```
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&consumeiterable_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_consumeiterable, consumeiterable_user_cmodule, MODULE_
↪CONSUMEITERABLE_ENABLED);
```

https://github.com/v923z/micropython-usermod/tree/master/snippets/consumeiterable/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/consumeiterable.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import consumeiterable

a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(a)
print(consumeiterable.sumsq(a))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
385.0
```

9.2 Returning iterables

Let us suppose that the result of some operation is an iterable, e.g., a tuple, or a list. How would we return such an object? How about a function that returns the powers of its argument? In python

```
def powerit(base, exponent):
    return [base**e for e in range(0, exponent+1)]

powerit(2, 10)
```

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

and in C,

https://github.com/v923z/micropython-usermod/tree/master/snippets/returniterable/returniterable.c

```
#include "py/obj.h"
#include "py/runtime.h"

STATIC mp_obj_t powers_iterable(mp_obj_t base, mp_obj_t exponent) {
    int e = mp_obj_get_int(exponent);
    mp_obj_t tuple[e+1];
```

(continues on next page)

(continued from previous page)

```
    int b = mp_obj_get_int(base), ba = 1;
    for(int i=0; i <= e; i++) {
        tuple[i] = mp_obj_new_int(ba);
        ba *= b;
    }
    return mp_obj_new_tuple(e+1, tuple);
}

STATIC MP_DEFINE_CONST_FUN_OBJ_2(powers_iterable_obj, powers_iterable);

STATIC const mp_rom_map_elem_t returniterable_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_returniterable) },
    { MP_ROM_QSTR(MP_QSTR_powers), MP_ROM_PTR(&powers_iterable_obj) },
};
STATIC MP_DEFINE_CONST_DICT(returniterable_module_globals, returniterable_module_
↪globals_table);

const mp_obj_module_t returniterable_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&returniterable_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_returniterable, returniterable_user_cmodule, MODULE_
↪RETURNITERABLE_ENABLED);
```

As everything else, the elements of tuples and lists are objects of type `mp_obj_t`, so, after finding out how far we have got to go with the exponents, we declare an array of the required length. Values are generated and assigned in the `for` loop. Since on the left hand side of the assignment we have an `mp_obj_t`, we convert the results with `mp_obj_new_int`. Once we are done with the computations, we return the array with `mp_obj_new_tuple`. This functions takes the array as the second argument, while the first argument specifies the length.

If you happen to want to return a list instead of a tuple, all you have to do is use `mp_obj_new_list` instead at the very end.

https://github.com/v923z/micropython-usermod/tree/master/snippets/returniterable/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/returniterable.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import returniterable
print(returniterable.powers(3, 10))
```

```
(1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049)
```

9.3 Creating iterables

Having seen how we can consume the elements in an iterable, it is time to explore what this `.getiter` magic is doing. So, let us create a new type, `itarray`, and make it iterable! This new type will have a constructor method,`square`, generating 16-bit integers, where the values are simply the squares of the indices, i.e., 1, 4, 9, 16... We are interested only in the iterability of the object, and for this reason, we will implement only the `.getiter` special method, and skip `.binary_op`, and `.unary_op`. If needed, these can easily be added based on the discussion in Special methods of classes.

Before listing the complete code, we discuss the relevant code snippets. The first chunk is the assignment of `.getiter` in the `iterable_array_type` structure. `.getiter` will be made equal to a function called `iterarray_getiter`, which simply returns `mp_obj_new_itarray_iterator`. Why can't we simply assign `mp_obj_new_itarray_iterator`, instead of wrapping it in `iterarray_getiter`? The reason for that is that `iterarray_getiter` has a strict signature, and we want to pass an extra argument, 0. This is nothing but the very first index in the sequence.

```
STATIC mp_obj_t itarray_getiter(mp_obj_t o_in, mp_obj_iter_buf_t *iter_buf) {
    return mp_obj_new_itarray_iterator(o_in, 0, iter_buf);
}

const mp_obj_type_t iterable_array_type = {
    { &mp_type_type },
    .name = MP_QSTR_itarray,
    .print = itarray_print,
    .make_new = itarray_make_new,
    .getiter = itarray_getiter,
};
```

So, it appears that we have to scrutinise `mp_obj_new_itarray_iterator`. This is a special object type in micropython, with a base type of `mp_type_polymorph_iter`. In addition, it holds a pointer to the `__next__` method, which is `itarray_iternext` in this case, stores a pointer to the variable (the one that we are iterating over), and the current index (which we initialised to 0 in `mp_obj_new_itarray_iterator`).

```
mp_obj_t mp_obj_new_itarray_iterator(mp_obj_t itarray, size_t cur, mp_obj_iter_buf_t␣
↪*iter_buf) {
    assert(sizeof(mp_obj_itarray_it_t) <= sizeof(mp_obj_iter_buf_t));
    mp_obj_itarray_it_t *o = (mp_obj_itarray_it_t*)iter_buf;
    o->base.type = &mp_type_polymorph_iter;
    o->iternext = itarray_iternext;
    o->itarray = itarray;
    o->cur = cur;
    return MP_OBJ_FROM_PTR(o);
}
```

`mp_obj_new_itarray_iterator` is not much more than a declaration and assignments. The object that we return is of type `mp_obj_itarray_it_t`, which has the above-mentioned structure

```
// itarray iterator
typedef struct _mp_obj_itarray_it_t {
    mp_obj_base_t base;
    mp_fun_1_t iternext;
    mp_obj_t itarray;
    size_t cur;
} mp_obj_itarray_it_t;

mp_obj_t itarray_iternext(mp_obj_t self_in) {
```

(continues on next page)

(continued from previous page)

```
    mp_obj_itarray_it_t *self = MP_OBJ_TO_PTR(self_in);
    itarray_obj_t *itarray = MP_OBJ_TO_PTR(self->itarray);
    if (self->cur < itarray->len) {
        // read the current value
        uint16_t *arr = itarray->elements;
        mp_obj_t o_out = MP_OBJ_NEW_SMALL_INT(arr[self->cur]);
        self->cur += 1;
        return o_out;
    } else {
        return MP_OBJ_STOP_ITERATION;
    }
}
```

Now, the complete code in one chunk:

https://github.com/v923z/micropython-usermod/tree/master/snippets/makeiterable/makeiterable.c

```
#include <stdlib.h>
#include "py/obj.h"
#include "py/runtime.h"

typedef struct _itarray_obj_t {
    mp_obj_base_t base;
    mp_fun_1_t iternext;
    uint16_t *elements;
    size_t len;
} itarray_obj_t;

const mp_obj_type_t iterable_array_type;
mp_obj_t mp_obj_new_itarray_iterator(mp_obj_t , size_t , mp_obj_iter_buf_t *);

STATIC void itarray_print(const mp_print_t *print, mp_obj_t self_in, mp_print_kind_t␣
↪kind) {
    (void)kind;
    itarray_obj_t *self = MP_OBJ_TO_PTR(self_in);
    printf("itarray: ");
    for(uint16_t i=0; i < self->len; i++) {
        printf("%d ", self->elements[i]);
    }
    printf("\n");
}

STATIC mp_obj_t itarray_make_new(const mp_obj_type_t *type, size_t n_args, size_t n_
↪kw, const mp_obj_t *args) {
    mp_arg_check_num(n_args, n_kw, 1, 1, true);
    itarray_obj_t *self = m_new_obj(itarray_obj_t);
    self->base.type = &iterable_array_type;
    self->len = mp_obj_get_int(args[0]);
    uint16_t *arr = malloc(self->len * sizeof(uint16_t));
    for(uint16_t i=0; i < self->len; i++) {
        arr[i] = i*i;
    }
    self->elements = arr;
    return MP_OBJ_FROM_PTR(self);
}

STATIC mp_obj_t itarray_getiter(mp_obj_t o_in, mp_obj_iter_buf_t *iter_buf) {
```

(continues on next page)

(continued from previous page)

```
    return mp_obj_new_itarray_iterator(o_in, 0, iter_buf);
}

const mp_obj_type_t iterable_array_type = {
    { &mp_type_type },
    .name = MP_QSTR_itarray,
    .print = itarray_print,
    .make_new = itarray_make_new,
    .getiter = itarray_getiter,
};

STATIC const mp_rom_map_elem_t makeiterable_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_makeiterable) },
    { MP_OBJ_NEW_QSTR(MP_QSTR_square), (mp_obj_t)&iterable_array_type },
};
STATIC MP_DEFINE_CONST_DICT(makeiterable_module_globals, makeiterable_module_globals_
↪table);

const mp_obj_module_t makeiterable_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&makeiterable_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_makeiterable, makeiterable_user_cmodule, MODULE_
↪MAKEITERABLE_ENABLED);

// itarray iterator
typedef struct _mp_obj_itarray_it_t {
    mp_obj_base_t base;
    mp_fun_1_t iternext;
    mp_obj_t itarray;
    size_t cur;
} mp_obj_itarray_it_t;

mp_obj_t itarray_iternext(mp_obj_t self_in) {
    mp_obj_itarray_it_t *self = MP_OBJ_TO_PTR(self_in);
    itarray_obj_t *itarray = MP_OBJ_TO_PTR(self->itarray);
    if (self->cur < itarray->len) {
        // read the current value
        uint16_t *arr = itarray->elements;
        mp_obj_t o_out = MP_OBJ_NEW_SMALL_INT(arr[self->cur]);
        self->cur += 1;
        return o_out;
    } else {
        return MP_OBJ_STOP_ITERATION;
    }
}

mp_obj_t mp_obj_new_itarray_iterator(mp_obj_t itarray, size_t cur, mp_obj_iter_buf_t␣
↪*iter_buf) {
    assert(sizeof(mp_obj_itarray_it_t) <= sizeof(mp_obj_iter_buf_t));
    mp_obj_itarray_it_t *o = (mp_obj_itarray_it_t*)iter_buf;
    o->base.type = &mp_type_polymorph_iter;
    o->iternext = itarray_iternext;
    o->itarray = itarray;
    o->cur = cur;
    return MP_OBJ_FROM_PTR(o);
```

(continues on next page)

(continued from previous page)

```
}
```

```
%%micropython

import makeiterable

a = makeiterable.square(15)
print(a)
for j, i in enumerate(a):
    if j == 1: print('%dst element: %d'%(j, i))
    elif j == 2: print('%dnd element: %d'%(j, i))
    elif j == 3: print('%drd element: %d'%(j, i))
    else:
        print('%dth element: %d'%(j, i))
```

```
itarray: 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196

0th element: 0
1st element: 1
2nd element: 4
3rd element: 9
4th element: 16
5th element: 25
6th element: 36
7th element: 49
8th element: 64
9th element: 81
10th element: 100
11th element: 121
12th element: 144
13th element: 169
14th element: 196
```

9.4 Subscripts

We now know, how we construct something that can be passed to a `for` loop. This is a good start. But iterables have other very useful properties. For instance, have you ever wondered, what actually happens in the following snippet?

```
a = 'micropython'
a[5]
```

```
'p'
```

`a` is a string, therefore, an iterable. Where does the interpreter know from, that it has got to return `p`, when asked for `a[5]`? Or have you ever been curious to know, how the interpreter replaces `p` by `q`, if

```
a = [c for c in 'micropyton']
a[5] = 'q'
a
```

```
['m', 'i', 'c', 'r', 'o', 'q', 'y', 't', 'o', 'n']
```

is passed to it? If so, then it is your lucky day, because we are going to make our iterable class be able to deal with such requests.

The code snippets above rely on a single special method, the subscription. In the C code of micropython, this method is called `.subscr`, and it should be assigned to in the class declaration, i.e., if we take `makeiterable.c` as our basis for the following discussion, then we would have to extend the `iterable_array_type` as

```
const mp_obj_type_t iterable_array_type {
    ...
    .subscr = itarray_subscr
}
```

where the signature of `itarray_subscr` has the form

```
STATIC mp_obj_t itarray_subscr(mp_obj_t self_in, mp_obj_t index, mp_obj_t value)
```

If `.subscr` is not implemented, but you are daring enough to call

```
>>> a[5]
```

all the same, then the interpreter is going to throw a `TypeError`:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'itarray' object isn't subscriptable
```

So, what happens in the method that we assigned in `iterable_array_type`? A possible scenario is given below:

```
STATIC mp_obj_t subitarray_subscr(mp_obj_t self_in, mp_obj_t index, mp_obj_t value) {
    subitarray_obj_t *self = MP_OBJ_TO_PTR(self_in);
    size_t idx = mp_obj_get_int(index);
    if(self->len <= idx) {
        mp_raise_ValueError("index is out of range");
    }
    if (value == MP_OBJ_SENTINEL) { // simply return the value at index, no assignment
        return MP_OBJ_NEW_SMALL_INT(self->elements[idx]);
    } else { // value was passed, replace the element at index
        self->elements[idx] = mp_obj_get_int(value);
    }
    return mp_const_none;
}
```

`subitarray_subscr` takes three arguments: the first is the instance on which the method is called, i.e., `self`. The second is the index, i.e., what stands in []. And finally, the third argument is the value. This is what we assign to the element at index `idx`, or, when we do not assign anything (i.e., when we *load* a value from the iterable), then `value` takes on a special value. If we have

```
>>> a[5]
```

on the python console, then the interpreter will automatically assign `value = MP_OBJ_SENTINEL` (this is defined in `obj.h`), so that, though we did not explicitly set anything to it, we can still inspect `value`. This is what happens, when we evaluate `value == MP_OBJ_SENTINEL`: if this statement is true, then we query for `a[5]`. Note that we also implemented some very rudimentary error checking: we raise an `IndexError`, whenever the index is out of range. We do this by calling

```
mp_raise_msg(&mp_type_IndexError, "index is out of range");
```

For a thorough discussion on how to raise exceptions see the Section *Error handling*.

There is one more thing that we should notice: at the very beginning of the function, in the line

```
size_t idx = mp_obj_get_int(index);
```

we call `mp_obj_get_int`. This means that any python object with an integer value is a valid argument, i.e., the following instruction would still work

```
%%micropython

a = 'micropython'
b = 5
print(a[b])
```

```
p
```

For compiling, here is the complete code:

https://github.com/v923z/micropython-usermod/tree/master/snippets/subscriptiterable/subscriptiterable.c

```
#include <stdlib.h>
#include "py/obj.h"
#include "py/runtime.h"

typedef struct _subitarray_obj_t {
    mp_obj_base_t base;
    mp_fun_1_t iternext;
    uint16_t *elements;
    size_t len;
} subitarray_obj_t;

const mp_obj_type_t subiterable_array_type;
mp_obj_t mp_obj_new_subitarray_iterator(mp_obj_t , size_t , mp_obj_iter_buf_t *);

STATIC void subitarray_print(const mp_print_t *print, mp_obj_t self_in, mp_print_kind_
↪t kind) {
    (void)kind;
    subitarray_obj_t *self = MP_OBJ_TO_PTR(self_in);
    printf("subitarray: ");
    for(uint16_t i=0; i < self->len; i++) {
        printf("%d ", self->elements[i]);
    }
    printf("\n");
}

STATIC mp_obj_t subitarray_make_new(const mp_obj_type_t *type, size_t n_args, size_t␣
↪n_kw, const mp_obj_t *args) {
    mp_arg_check_num(n_args, n_kw, 1, 1, true);
    subitarray_obj_t *self = m_new_obj(subitarray_obj_t);
    self->base.type = &subiterable_array_type;
    self->len = mp_obj_get_int(args[0]);
    uint16_t *arr = malloc(self->len * sizeof(uint16_t));
    for(uint16_t i=0; i < self->len; i++) {
        arr[i] = i*i;
    }
    self->elements = arr;
    return MP_OBJ_FROM_PTR(self);
}
```

(continues on next page)

(continued from previous page)

```
STATIC mp_obj_t subitarray_getiter(mp_obj_t o_in, mp_obj_iter_buf_t *iter_buf) {
    return mp_obj_new_subitarray_iterator(o_in, 0, iter_buf);
}

STATIC mp_obj_t subitarray_subscr(mp_obj_t self_in, mp_obj_t index, mp_obj_t value) {
    subitarray_obj_t *self = MP_OBJ_TO_PTR(self_in);
    size_t idx = mp_obj_get_int(index);
    if(self->len <= idx) {
        mp_raise_msg(&mp_type_IndexError, "index is out of range");
    }
    if (value == MP_OBJ_SENTINEL) { // simply return the value at index, no assignment
        return MP_OBJ_NEW_SMALL_INT(self->elements[idx]);
    } else { // value was passed, replace the element at index
        self->elements[idx] = mp_obj_get_int(value);
    }
    return mp_const_none;
}

const mp_obj_type_t subiterable_array_type = {
    { &mp_type_type },
    .name = MP_QSTR_subitarray,
    .print = subitarray_print,
    .make_new = subitarray_make_new,
    .getiter = subitarray_getiter,
    .subscr = subitarray_subscr,
};

STATIC const mp_rom_map_elem_t subscriptiterable_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_subscriptiterable) },
    { MP_OBJ_NEW_QSTR(MP_QSTR_square), (mp_obj_t)&subiterable_array_type },
};
STATIC MP_DEFINE_CONST_DICT(subscriptiterable_module_globals, subscriptiterable_
↪module_globals_table);

const mp_obj_module_t subscriptiterable_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&subscriptiterable_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_subscriptiterable, subscriptiterable_user_cmodule, MODULE_
↪SUBSCRIPTITERABLE_ENABLED);

// itarray iterator
typedef struct _mp_obj_subitarray_it_t {
    mp_obj_base_t base;
    mp_fun_1_t iternext;
    mp_obj_t subitarray;
    size_t cur;
} mp_obj_subitarray_it_t;

mp_obj_t subitarray_iternext(mp_obj_t self_in) {
    mp_obj_subitarray_it_t *self = MP_OBJ_TO_PTR(self_in);
    subitarray_obj_t *subitarray = MP_OBJ_TO_PTR(self->subitarray);
    if (self->cur < subitarray->len) {
        // read the current value
        uint16_t *arr = subitarray->elements;
        mp_obj_t o_out = MP_OBJ_NEW_SMALL_INT(arr[self->cur]);
```

(continues on next page)

(continued from previous page)

```
        self->cur += 1;
        return o_out;
    } else {
        return MP_OBJ_STOP_ITERATION;
    }
}

mp_obj_t mp_obj_new_subitarray_iterator(mp_obj_t subitarray, size_t cur, mp_obj_iter_
↪buf_t *iter_buf) {
    assert(sizeof(mp_obj_subitarray_it_t) <= sizeof(mp_obj_iter_buf_t));
    mp_obj_subitarray_it_t *o = (mp_obj_subitarray_it_t*)iter_buf;
    o->base.type = &mp_type_polymorph_iter;
    o->iternext = subitarray_iternext;
    o->subitarray = subitarray;
    o->cur = cur;
    return MP_OBJ_FROM_PTR(o);
}
```

https://github.com/v923z/micropython-usermod/tree/master/snippets/subscriptiterable/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/subscriptiterable.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import subscriptiterable
a = subscriptiterable.square(15)
print(a)
print('the third element is %d'%a[3])
b = 3+7
a[b] = 0
print(a)
```

```
subitarray: 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196

the third element is 9
subitarray: 0 1 4 9 16 25 36 49 64 81 0 121 144 169 196
```

9.4.1 Index reversing

Now, the code above works for non-negative indices, but in python it is quite customary to have something like

```
a = 'micropython'
a[-2]
```

```
'o'
```

which is equivalent to querying for the last but one element (second from the right) in the iterable. Knowing how long the iterable is (this is stored in `self->len`), it is a trivial matter to modify our code in such a way that it can return the values at negative indices.

9.5 Slicing

In the previous two sections we have worked with single elements of an iterable. But python wouldn't be python without slices. Slices are index ranges specified in a `start:end:step` format. Taking our earlier example, we can print every second character in `micropython` by

```
a = 'micropython'
a[0:8:2]
```

```
'mcoy'
```

This behaviour is also part of the `.subscr` special method. Let us implement it, shall we? In order to simplify the discussion, we will treat one case only: returning values, and we return a new instance of the array, if a slice was requested, while a single number, if we passed a single index.

Since we want to return an array if the indices stem from a slice, we split our original `subscriptitarray_make_new` function, and separate those parts that reserve space for the array from those that do the assignments.

It shouldn't come as a surprise that we have to modify the function that was hooked up to `.subscr`. Let us take a look at the following snippet:

```
STATIC mp_obj_t sliceitarray_subscr(mp_obj_t self_in, mp_obj_t index, mp_obj_t value)
↪{
    sliceitarray_obj_t *self = MP_OBJ_TO_PTR(self_in);
    if (value == MP_OBJ_SENTINEL) { // simply return the values at index, no␣
↪assignment

#if MICROPY_PY_BUILTINS_SLICE
        if (MP_OBJ_IS_TYPE(index, &mp_type_slice)) {
            mp_bound_slice_t slice;
            mp_seq_get_fast_slice_indexes(self->len, index, &slice);
            uint16_t len = (slice.stop - slice.start) / slice.step;
            sliceitarray_obj_t *res = create_new_sliceitarray(len);
            for(size_t i=0; i < len; i++) {
                res->elements[i] = self->elements[slice.start+i*slice.step];
            }
            return MP_OBJ_FROM_PTR(res);
        }
#endif
        // we have a single index, return a single number
        size_t idx = mp_obj_get_int(index);
        return MP_OBJ_NEW_SMALL_INT(self->elements[idx]);
    } else { // do not deal with assignment, bail out
        return mp_const_none;
    }
    return mp_const_none;
}
```

As advertised, we treat only the case, when `value` is empty, i.e., it is equal to an `MP_OBJ_SENTINEL`. Now, there is no point in trying to read out the parameters of a slice, if the slice object is not even defined, is there? This is the case for the minimal ports. So, in order to prevent nasty things from happening, we insert the `#if/#endif` macro with the parameter `MICROPY_PY_BUILTINS_SLICE`. Provided that `MICROPY_PY_BUILTINS_SLICE` is defined, we inspect the index, and find out if it is a slice by calling

```
MP_OBJ_IS_TYPE(index, &mp_type_slice)
```

If so, we attempt to load the slice parameters into the `slice` object with

```
mp_seq_get_fast_slice_indexes(self->len, index, &slice)
```

The function `mp_seq_get_fast_slice_indexes` returns Boolean `true`, if the increment in the slice is 1, and `false` otherwise. For the goal that we are trying to pursue here, it doesn't matter what the step size is, so we don't care about the return value. But the main purpose of the function is actually something different: the function expands the `start:end:step` slice into a triplet, and it does so, even if one or two of the slice parameters are missing. So, `start::step`, `start::`, `:end:step` etc. will also work. In fact, this is why we have to pass the length of the array: `self->len` will be substituted, if the `:end:` parameter is missing.

Equipped with the values of `slice.start`, `slice.stop`, and `slice.step`, we can determine the length of the new array, and assign the values in the `for` loop.

https://github.com/v923z/micropython-usermod/tree/master/snippets/sliceiterable/sliceiterable.c

```
#include <stdlib.h>
#include "py/obj.h"
#include "py/runtime.h"

typedef struct _sliceitarray_obj_t {
    mp_obj_base_t base;
    mp_fun_1_t iternext;
    uint16_t *elements;
    size_t len;
} sliceitarray_obj_t;

const mp_obj_type_t sliceiterable_array_type;
mp_obj_t mp_obj_new_sliceitarray_iterator(mp_obj_t , size_t , mp_obj_iter_buf_t *);

STATIC void sliceitarray_print(const mp_print_t *print, mp_obj_t self_in, mp_print_
↪kind_t kind) {
    (void)kind;
    sliceitarray_obj_t *self = MP_OBJ_TO_PTR(self_in);
    printf("sliceitarray: ");
    for(uint16_t i=0; i < self->len; i++) {
        printf("%d ", self->elements[i]);
    }
    printf("\n");
}

sliceitarray_obj_t *create_new_sliceitarray(uint16_t len) {
    sliceitarray_obj_t *self = m_new_obj(sliceitarray_obj_t);
    self->base.type = &sliceiterable_array_type;
    self->len = len;
    uint16_t *arr = malloc(self->len * sizeof(uint16_t));
    self->elements = arr;
    return self;
}
```

(continues on next page)

(continued from previous page)

```
STATIC mp_obj_t sliceitarray_make_new(const mp_obj_type_t *type, size_t n_args, size_
↪t n_kw, const mp_obj_t *args) {
    mp_arg_check_num(n_args, n_kw, 1, 1, true);
    sliceitarray_obj_t *self = create_new_sliceitarray(mp_obj_get_int(args[0]));
    for(uint16_t i=0; i < self->len; i++) {
        self->elements[i] = i*i;
    }
    return MP_OBJ_FROM_PTR(self);
}

STATIC mp_obj_t sliceitarray_getiter(mp_obj_t o_in, mp_obj_iter_buf_t *iter_buf) {
    return mp_obj_new_sliceitarray_iterator(o_in, 0, iter_buf);
}

STATIC mp_obj_t sliceitarray_subscr(mp_obj_t self_in, mp_obj_t index, mp_obj_t value)
↪{
    sliceitarray_obj_t *self = MP_OBJ_TO_PTR(self_in);
    if (value == MP_OBJ_SENTINEL) { // simply return the values at index, no␣
↪assignment

#if MICROPY_PY_BUILTINS_SLICE
        if (MP_OBJ_IS_TYPE(index, &mp_type_slice)) {
            mp_bound_slice_t slice;
            mp_seq_get_fast_slice_indexes(self->len, index, &slice);
            printf("start: %ld, stop: %ld, step: %ld\n", slice.start, slice.stop,␣
↪slice.step);
            uint16_t len = (slice.stop - slice.start) / slice.step;
            sliceitarray_obj_t *res = create_new_sliceitarray(len);
            for(size_t i=0; i < len; i++) {
                res->elements[i] = self->elements[slice.start+i*slice.step];
            }
            return MP_OBJ_FROM_PTR(res);
        }
#endif
        // we have a single index, return a single number
        size_t idx = mp_obj_get_int(index);
        return MP_OBJ_NEW_SMALL_INT(self->elements[idx]);
    } else { // do not deal with assignment, bail out
        return mp_const_none;
    }
    return mp_const_none;
}

const mp_obj_type_t sliceiterable_array_type = {
    { &mp_type_type },
    .name = MP_QSTR_sliceitarray,
    .print = sliceitarray_print,
    .make_new = sliceitarray_make_new,
    .getiter = sliceitarray_getiter,
    .subscr = sliceitarray_subscr,
};

STATIC const mp_rom_map_elem_t sliceiterable_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_sliceiterable) },
    { MP_OBJ_NEW_QSTR(MP_QSTR_square), (mp_obj_t)&sliceiterable_array_type },
};
STATIC MP_DEFINE_CONST_DICT(sliceiterable_module_globals, sliceiterable_module_
↪globals_table);
```

(continues on next page)

(continued from previous page)

```
const mp_obj_module_t sliceiterable_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&sliceiterable_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_sliceiterable, sliceiterable_user_cmodule, MODULE_
↪SLICEITERABLE_ENABLED);

// itarray iterator
typedef struct _mp_obj_sliceitarray_it_t {
    mp_obj_base_t base;
    mp_fun_1_t iternext;
    mp_obj_t sliceitarray;
    size_t cur;
} mp_obj_sliceitarray_it_t;

mp_obj_t sliceitarray_iternext(mp_obj_t self_in) {
    mp_obj_sliceitarray_it_t *self = MP_OBJ_TO_PTR(self_in);
    sliceitarray_obj_t *sliceitarray = MP_OBJ_TO_PTR(self->sliceitarray);
    if (self->cur < sliceitarray->len) {
        // read the current value
        uint16_t *arr = sliceitarray->elements;
        mp_obj_t o_out = MP_OBJ_NEW_SMALL_INT(arr[self->cur]);
        self->cur += 1;
        return o_out;
    } else {
        return MP_OBJ_STOP_ITERATION;
    }
}

mp_obj_t mp_obj_new_sliceitarray_iterator(mp_obj_t sliceitarray, size_t cur, mp_obj_
↪iter_buf_t *iter_buf) {
    assert(sizeof(mp_obj_sliceitarray_it_t) <= sizeof(mp_obj_iter_buf_t));
    mp_obj_sliceitarray_it_t *o = (mp_obj_sliceitarray_it_t*)iter_buf;
    o->base.type = &mp_type_polymorph_iter;
    o->iternext = sliceitarray_iternext;
    o->sliceitarray = sliceitarray;
    o->cur = cur;
    return MP_OBJ_FROM_PTR(o);
}
```

https://github.com/v923z/micropython-usermod/tree/master/snippets/sliceiterable/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/sliceiterable.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import sliceiterable
a = sliceiterable.square(20)

print(a)
print(a[1:15:3])
```

```
sliceitarray: 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361

start: 1, stop: 15, step: 3
sliceitarray: 1 16 49 100
```

A word of caution is in order here: if the step size is negative, the array is reversed. This means that `slice.start` is larger than `slice.stop`, and when we calculate the length of the new array, we end up with a negative number. Just saying.

CHAPTER 10

Profiling

There are times, when you might want to find out what resources (time and RAM) a particular operation requires. Not because you are nosy, but because the resources of a microcontroller are limited, therefore, if you are out of luck, the desired operation might not even fit within the constraints of the chip. In order to locate the bottleneck, you will need to do a bit of profiling. Or perhaps, a lot. This is what we are going to discuss now.

Since you are not going to face serious difficulties when running micropython on a computer, profiling makes really sense only in the context of the microcontroller, so this might be a golden opportunity to brush up on how the firmware has to be compiled and uploaded. It is not by accident that we spent some time on this at the very beginning of this document.

10.1 Profiling in python

10.1.1 Measuring time

If you are interested in the execution time of a complete function, you can measure it simply by making use of the python interpreter

```
%%micropython

from utime import ticks_us, ticks_diff

def test_function(n):
    for i in range(n):
        q = i*i*i
    return q # return the last

now = ticks_us()
test_function(100)
then = ticks_diff(ticks_us(), now)

print("function test_function() took %d us to run"%then)
```

```
function test_function() took 27 us to run
```

In fact, since our function is flanked by two other statements, this construct easily lends itself to a decorator implementation, as in (taken from http://docs.micropython.org/en/v1.9.3/pyboard/reference/speed_python.html)

```
%%micropython

import utime

def timed_function(f, *args, **kwargs):
    myname = str(f).split(' ')[1]
    def new_func(*args, **kwargs):
        t = utime.ticks_us()
        result = f(*args, **kwargs)
        delta = utime.ticks_diff(utime.ticks_us(), t)
        print('Function {} time = {:6.3f}ms'.format(myname, delta/1000))
        return result
    return new_func

@timed_function
def test_function(n):
    for i in range(n):
        utime.sleep_ms(10)

test_function(10)
```

```
Function test_function time = 100.682ms
```

(If you need an even better estimate, you can get the ticks twice, and yank `run_my_function()` in the second pass: in this way, you would get the cost of measuring time itself:

```
from utime import ticks_us, ticks_diff

now = ticks_us
then = ticks_diff(ticks_us(), now)

print("the time measurement took %d us"%then)
```

Then you subtract the results of the second measurement from those of the first.)

10.1.2 The memory cost of a function

While time is money, RAM is gold. We shouldn't pass up on that! The `micropython` has a very handy function for printing a pretty summary of the state of the RAM. You would call it like

```
%%micropython

import micropython
print(micropython.mem_info())
```

```
mem: total=2755, current=663, peak=2289
stack: 928 out of 80000
GC: total: 2072832, used: 704, free: 2072128
 No. of 1-blocks: 6, 2-blocks: 3, max blk sz: 6, max free sz: 64745
None
```

If you call `mem_info()` after you executed your function, but before calling the garbage collector (if that is enabled, that is), then from the two reports, you can figure out how many bytes the function has eaten.

10.2 Profiling in C

With the profiling method above, we can measure the cost of a complete function only, but we cannot say anything about individual instructions in the body. Execution time is definitely a significant issue, but even worse is the problem of RAM: it might happen that the function allocates a huge amount of memory, but cleans up properly before returning. Such a function could certainly wreak havoc, even if it is rather inocuous-looking from the outside. So, what do we do? We should probably just measure. It is not going to hurt.

In the example below (`profiling.c`), I discuss both time and RAM measurements in a single module, because splitting them wouldn't be worth the trouble. The function, whose behaviour we inspect, does nothing, but calculate the length of a three-dimensional vector. With that, we can figure out, how much the assignment, and how much the actual calculation cost.

https://github.com/v923z/micropython-usermod/tree/master/snippets/profiling/profiling.c

```
#include <math.h>
#include <stdio.h>
#include "py/obj.h"
#include "py/runtime.h"
#include "mphalport.h"  // needed for mp_hal_ticks_cpu()
#include "py/builtin.h" // needed for mp_micropython_mem_info()

STATIC mp_obj_t measure_cpu(mp_obj_t _x, mp_obj_t _y, mp_obj_t _z) {
    size_t start, middle, end;
    start = m_get_current_bytes_allocated();

    float x = mp_obj_get_float(_x);
    float y = mp_obj_get_float(_y);
    float z = mp_obj_get_float(_z);
    middle = m_get_current_bytes_allocated();

    float hypo = sqrtf(x*x + y*y + z*z);
    end = m_get_current_bytes_allocated();
    mp_obj_t tuple[4];
    tuple[0] = MP_OBJ_NEW_SMALL_INT(start);
    tuple[1] = MP_OBJ_NEW_SMALL_INT(middle);
    tuple[2] = MP_OBJ_NEW_SMALL_INT(end);
    tuple[3] = mp_obj_new_float(hypo);
    return mp_obj_new_tuple(4, tuple);
}

STATIC MP_DEFINE_CONST_FUN_OBJ_3(measure_cpu_obj, measure_cpu);

STATIC const mp_rom_map_elem_t profiling_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_profiling) },
    { MP_ROM_QSTR(MP_QSTR_measure), MP_ROM_PTR(&measure_cpu_obj) },
};
STATIC MP_DEFINE_CONST_DICT(profiling_module_globals, profiling_module_globals_table);

const mp_obj_module_t profiling_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&profiling_module_globals,
};
```

(continues on next page)

(continued from previous page)

```
MP_REGISTER_MODULE(MP_QSTR_profiling, profiling_user_cmodule, MODULE_PROFILING_
↪ENABLED);
```

The above-mentioned `mem_info()` function of the micropython module can directly be called from C: after including the `builtin.h` header, we can issue `mp_micropython_mem_info(0, NULL);`, defined in `modmicropython.c`, which will print everything we need. Although its signature contains two arguments, a `size_t` and an `mp_obj_t` pointer to the arguments, the function does not seem to care about them, so we can pass `0`, and `NULL` without any meaning.

The function `mp_micropython_mem_info()` doesn't carry out any measurements in itself, it is only for pretty printing. The stats are collected by `mp_micropython_mem_total()`, `mp_micropython_mem_current()`, and `mp_micropython_mem_peak()`. Unfortunately, these functions are all declared STATIC, so we cannot call them from outsize `modmicropython.c`. If you need a numeric representation of the state of the RAM, you can make use of the `m_get_total_bytes_allocated(void)`, `m_get_current_bytes_allocated(void)`, and `m_get_peak_bytes_allocated(void)` functions of `py/malloc.c`. All three return a `size_t`.

With the help of these three functions, we could, e.g., return the size of the consumed memory to the micropython interpreter at the end of our calculations. This is what we do, when collecting the bits an pieces, and returning the 4-tuple at the end of the `measure_cpu` function.

https://github.com/v923z/micropython-usermod/tree/master/snippets/profiling/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/profiling.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import profiling
print(profiling.measure(123, 233, 344))
```

```
(672, 672, 672, 433.305908203125)
```

CHAPTER 11

Working with larger modules

Once you add more and more functionality and functions to your module, it will become unmanageably, and it might make more sense to split the module into smaller components. We are going to hack our very first module, `simplefunction`, and factor out the function in it.

Since we will want to refer to our functions in the module definition, we have to declare them in a header file. Let us call this file `helper.h`. The functions declared therein operate on `micropython` types, so do not forget to include `py/obj.h`, and possibly `py/runtime.h`!

https://github.com/v923z/micropython-usermod/tree/master/snippets/largemodule/helper.h

```
#include "py/obj.h"
#include "py/runtime.h"

mp_obj_t largemodule_add_ints(mp_obj_t , mp_obj_t );
mp_obj_t largemodule_subtract_ints(mp_obj_t , mp_obj_t );
```

Next, in `helper.c`, we have to implement the functions. `helper.c` should also contain the declarations, i.e., `header.h` has to be included.

https://github.com/v923z/micropython-usermod/tree/master/snippets/largemodule/helper.c

```
#include "helper.h"

mp_obj_t largemodule_add_ints(mp_obj_t a_obj, mp_obj_t b_obj) {
    int a = mp_obj_get_int(a_obj);
    int b = mp_obj_get_int(b_obj);
    return mp_obj_new_int(a + b);
}

mp_obj_t largemodule_subtract_ints(mp_obj_t a_obj, mp_obj_t b_obj) {
    int a = mp_obj_get_int(a_obj);
    int b = mp_obj_get_int(b_obj);
    return mp_obj_new_int(a - b);
}
```

Finally, in the module implementation, we include `helper.h`, and create the function objects with `MP_DEFINE_CONST_FUN_OBJ_2`, and its relatives. The rest of the code is equivalent to `simplefunction.c`, with the only exception of the module name.

https://github.com/v923z/micropython-usermod/tree/master/snippets/largemodule/largemodule.c

```
#include "py/obj.h"
#include "py/runtime.h"
#include "helper.h"


STATIC MP_DEFINE_CONST_FUN_OBJ_2(largemodule_add_ints_obj, largemodule_add_ints);
STATIC MP_DEFINE_CONST_FUN_OBJ_2(largemodule_subtract_ints_obj, largemodule_subtract_
↪ints);

STATIC const mp_rom_map_elem_t largemodule_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_largemodule) },
    { MP_ROM_QSTR(MP_QSTR_add_ints), MP_ROM_PTR(&largemodule_add_ints_obj) },
    { MP_ROM_QSTR(MP_QSTR_subtract_ints), MP_ROM_PTR(&largemodule_subtract_ints_obj) }
↪,
};
STATIC MP_DEFINE_CONST_DICT(largemodule_module_globals, largemodule_module_globals_
↪table);

const mp_obj_module_t largemodule_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&largemodule_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_largemodule, largemodule_user_cmodule, MODULE_LARGEMODULE_
↪ENABLED);
```

Now, since we have multiple files in our module, we have to change the `makefile` accordingly, and before linking, we have to compile both `helper.c`, and `largemodule.c`, thus, we add `$(USERMODULES_DIR)/helper.c`, *and* `$(USERMODULES_DIR)/largemodule.c` to `SRC_USERMOD`.

https://github.com/v923z/micropython-usermod/tree/master/snippets/largemodule/micropython.mk

```
USERMODULES_DIR := $(USERMOD_DIR)

# Add all C files to SRC_USERMOD.
SRC_USERMOD += $(USERMODULES_DIR)/helper.c
SRC_USERMOD += $(USERMODULES_DIR)/largemodule.c

# We can add our module folder to include paths if needed
# This is not actually needed in this example.
CFLAGS_USERMOD += -I$(USERMODULES_DIR)
```

```
!make USER_C_MODULES=../../../usermod/snippets/ all
```

```
%%micropython

import largemodule

print(largemodule.add_ints(1, 2))
print(largemodule.subtract_ints(1, 2))
```

```
3
-1
```

CHAPTER 12

A word for the lazy

If you still find that coding in C is too cumbersome, you can try your hand at a stub generator, e.g., https://github.com/pazzarpj/micropython-ustubby, or https://gitlab.com/oliver.robson/mpy-c-stub-gen . These tools allow you to convert your python code into C code. Basically, they will produce a boilerplate file, which you can flesh out with the C implementation of the required functionality.

Looking at the usage examples, it is clear to me that one can save a lot of typing with these stub generators, but one will still need a basic understanding of how to work with the micropython C code.

CHAPTER 13

Outline of a math library

As I indicated at the very beginning, my main motivation for writing this document was that I wanted to have a reasonable programming manual for the development of a math library. Since I couldn't find any, I have turned the problem around, and written up, what I have learnt by developing the library. But the question is, what this library should achieve in the first place?

13.1 Requirements

Recently, I have run into some limitations with the micropython interpreter. These difficulties were related to both speed, and RAM. Therefore, I wanted to have something that can perform common mathematical calculations in a pythonic way, with little burden on the RAM, and possibly fast. On PCs, such a library is called `numpy`, and it felt only natural to me to implement those aspects of `numpy` that would find an applications in the context of data acquisition of moderate volume: after all, no matter what, the microcontroller is not going to produce or collect huge amounts of data, but it might still be useful to process these data within the constraints of the microcontroller. Due to the nature of the data that would be dealt with, one can work with a very limited subset of `numpy`.

Keeping these considerations in mind, I set my goals as follows:

- One should be able to vectorise standard mathematical functions, while these functions should still work for scalars, so

```
a = 1.0
sin(a)
```

and

```
a = [1.0, 2.0, 3.0]
sin(a)
```

should both be valid expressions.

- There should be a binary container, (`ndarray`) for numbers that are results of vectorised operations, and one should be able to initialise a container by passing arbitrary `iterables` to a constructor (see `sin([1, 2, 3])` above).

- The array should be iterable, so that we can turn it into lists, tuples, etc.
- The relevant binary operations should work on arrays as in `numpy`, that is, e.g.,

```
>>> a = ndarray([1, 2, 3, 4])
>>> (a + 1) + a*10
```

should evaluate to `ndarray([12, 23, 34, 45])`.

- 2D arrays (matrices) could be useful (see below), thus, the above-mentioned container should be able to store its `shape`.
- Having matrices, it is only natural to implement standard matrix operations (inversion, transposition etc.)
- These numerical arrays and matrices should have a reasonable visual representation (pretty printing)
- With the help of matrices, one can also think of polynomial fits to measurement data
- There should be an FFT routine that can work with linear arrays. I do not think that 2D transforms would be very useful for data that come from the ADC of the microcontroller, but being able to extract frequency components of 1D signals would be an asset.

And this is, how `ulab` was born. But that is another story, for another day https://github.com/v923z/micropython-ulab/.

CHAPTER 14

Indices and tables

- genindex
- modindex
- search