Writing fast and efficient MicroPython

Damien P. George (George Robotics)

PyCon AU, Sydney, 24th August 2018

WHAT IS MICROPYTHON?

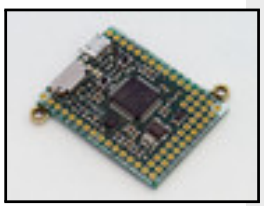
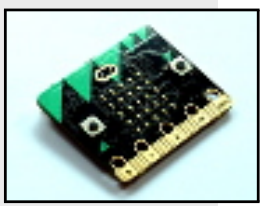
MicroPython is:

- a complete reimplementation of Python
- designed to be efficient with resources
- designed to run bare metal

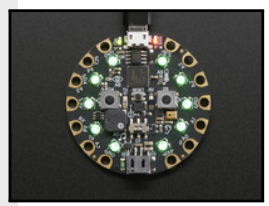
MicroPython includes:

- a compiler, runtime and familiar REPL
- support for basic libraries (modules), most begin with 'u'
- extra modules to control hardware

TRY IT OUT!





- download the firmware from `micropython.org/download`
- try it out online at `micropython.org/unicorn`

FAST AND EFFICIENT MICROPYTHON SCRIPTS

time – storage – energy use

Aim of this talk:

- give some insight to how MicroPython works internally
- based on insight, provide tips and tricks for efficiency
- work through some fun examples!

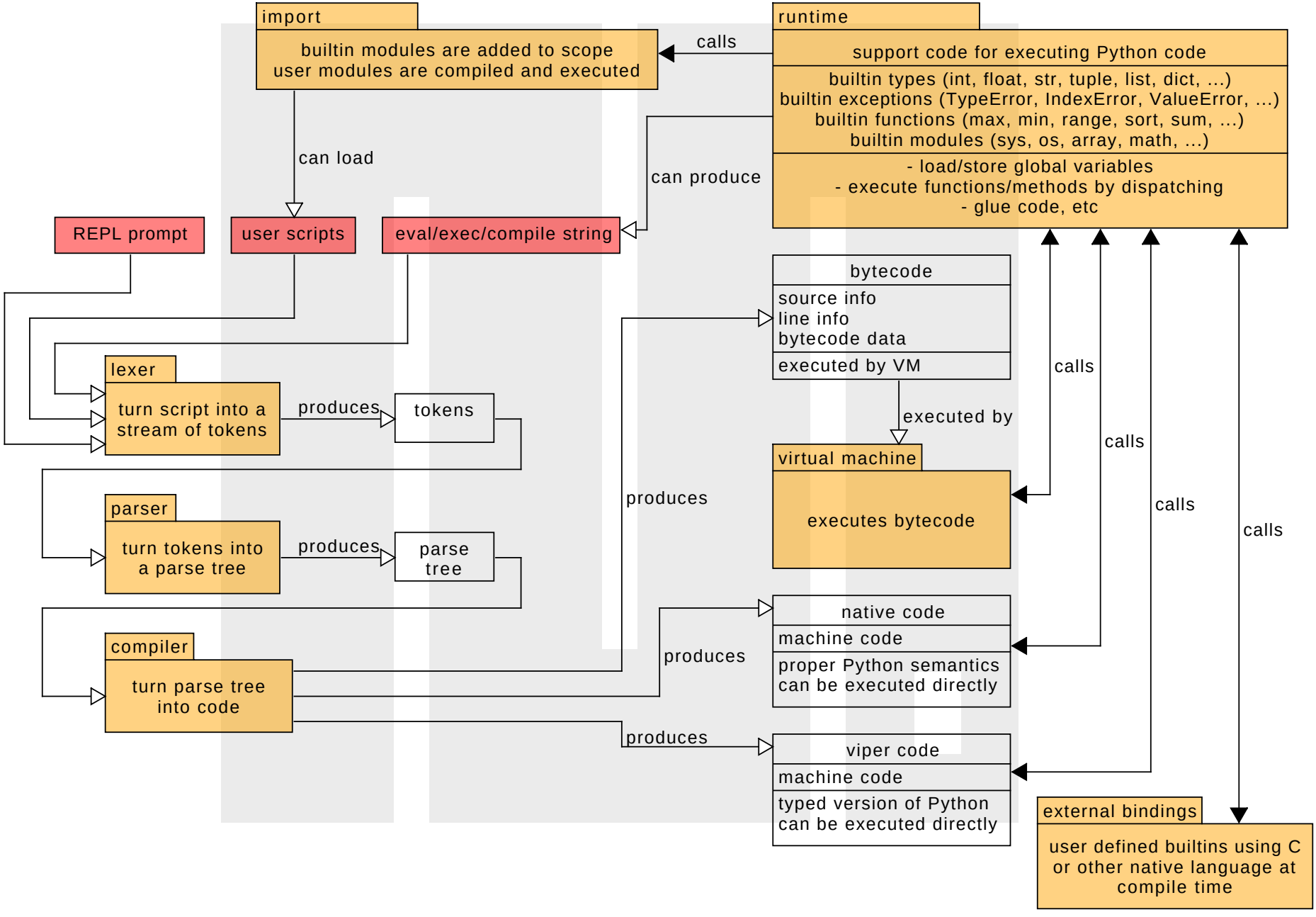
- **Blink LED: 50kHz too slow!**

```
led = machine.Pin('LED_BLUE')
for i in range(n):
    led.on()
    led.off()
```

- **Read data: 1.5MB/sec too slow!**

```
total = 0
with open(filename) as f:
    for i in range(n):
        total += len(f.read(1000))
        f.seek(0, 0)
return total
```

INTERNAL ARCHITECTURE



COMPILER

- reads input stream character-by-character
- can leave memory fragmented from parse tree
- all identifiers are left interned/stored in RAM

EMITTERS

- uses RAM to store generated code
- bytecode
- native machine code (x86, x64, ARM, Thumb, Xtensa)
- inline assembler

MICROPYTHON BYTECODES

```
LOAD_CONST_FALSE        LOAD_FAST_N          STORE_FAST_N      DELETE_FAST
LOAD_CONST_NONE         LOAD_DEREF           STORE_DEREF       DELETE_DERE
LOAD_CONST_TRUE         LOAD_NAME            STORE_NAME        DELETE_NAME
LOAD_CONST_SMALL_INT    LOAD_GLOBAL          STORE_GLOBAL      DELETE_GLOBAL
LOAD_CONST_STRING       LOAD_ATTR            STORE_ATTR
LOAD_CONST_OBJ          LOAD_METHOD          STORE_SUBSCR      GET_ITER
LOAD_NULL               LOAD_SUPER_METHOD                      GET_ITER_STACK
                        LOAD_BUILD_CLASS                       FOR_ITER
                        LOAD_SUBSCR

DUP_TOP                                      SETUP_WITH
DUP_TOP_TWO             JUMP                 WITH_CLEANUP      RETURN_VALUE
POP_TOP                 POP_JUMP_IF_TRUE     SETUP_EXCEPT      RAISE_VARARGS
ROT_TWO                 POP_JUMP_IF_FALSE    SETUP_FINALLY     YIELD_VALUE
ROT_THREE               JUMP_IF_TRUE_OR_POP  END_FINALLY       YIELD_FROM
                        JUMP_IF_FALSE_OR_POP POP_BLOCK
                        UNWIND_JUMP          POP_EXCEPT

                                                               IMPORT_NAME
BUILD_TUPLE               MAKE_FUNCTION                        IMPORT_FROM
BUILD_LIST                MAKE_FUNCTION_DEFARGS                IMPORT_STAR
BUILD_MAP                 MAKE_CLOSURE
BUILD_SET                 MAKE_CLOSURE_DEFARGS
BUILD_SLICE                                          LOAD_CONST_SMALL_INT_MULTI
STORE_MAP                 CALL_FUNCTION              LOAD_FAST_MULTI
STORE_COMP                CALL_FUNCTION_VAR_KW       STORE_FAST_MULTI
UNPACK_SEQUENCE           CALL_METHOD                UNARY_OP_MULTI
UNPACK_EX                 CALL_METHOD_VAR_KW         BINARY_OP_MULTI
```

EXAMPLE SCRIPT

```
def do_sleep(d):
    print("sleep", d)
    time.sleep(d)
```

```
00 LOAD_GLOBAL print
03 LOAD_CONST_STRING "sleep"
06 LOAD_FAST 0
07 CALL_FUNCTION n=2 nkw=0
09 POP_TOP
10 LOAD_GLOBAL time
13 LOAD_METHOD sleep
16 LOAD_FAST 0
17 CALL_METHOD n=1 nkw=0
19 POP_TOP
20 LOAD_CONST_NONE
21 RETURN_VALUE
```

MEMORY ALLOCATION

Many core constructs don't allocate on the heap:

- expressions
- `if`, `while`, `for` and `try` statements
- local variables
- small integer arithmetic
- inplace operations on *existing* data structures
- calling functions/methods with positional or keyword args
- some builtins: `all`, `any`, `callable`, `getattr`, `hasattr`, `isinstance`, `issubclass`, `len`, `max`, `min`, `ord`, `print`, `sum`

Common things that do allocate on the heap:

- importing
- defining functions and classes
- assigning global variables for the first time
- creating data structures

TIPS: CPU TIME

- use functions, not global scope
- use local variables
- cache module functions and object methods as locals
- cache self variables as locals
- prefer longer expressions, not split up ones
- runtime is faster than Python, use it; eg str.startswith
- `from micropython import const; X = const(1)`
- `1 << 3` is okay, will be optimised!

TIPS: RAM USAGE

- don't use heap when possible
- shorter variable names, reuse them; eg `x`, `y`, `i`, `len`, `var`
- temporary buffers: `self.buf1 = bytearray(1)`
- use `XXX_into` methods
- don't use `*` or `**` args
- `from micropython import const; _X = const(1)`
- script minification
- use `mpy-cross` to produce .mpy
- ultimate solution: freeze scripts into the firmware

- **Blink LED: 50kHz too slow, make it faster!**

```
led = machine.Pin('LED_BLUE')
for i in range(n):
    led.on()
    led.off()
```

- **Read data: 1.5MB/sec too slow, make it faster!**

```
1 total = 0
2 with open(filename) as f:
3     for i in range(n):
4         total += len(f.read(1000))
5         f.seek(0, 0)
6 return total
```

OTHER OPTIMISATIONS

- energy use: faster code can go to sleep longer
- programmer time
- debugging effort
- maintenance effort

SUMMARY

- *optimise at the end, only the things that are bottlenecks!*
- naive Python code roughly 100x slower than C
- BUT! can usually do a lot better
- use runtime functions/methods and C modules (eg `re`)
- use locals, preallocate memory, cache things

www.micropython.org

forum.micropython.org

github.com/micropython