Python
Bits and Bytes

Hamilton Python Users Group
Ian Stewart
2023-04-12

Python Meetings

Hamilton Python Users Group

First meeting Feb 2014.

Forgot to celebrate 100th meeting.

102nd Meeting Apr 2023.

Python

Designed by Guido van Rossum – The Netherlands
First appeared 20 February 1991; 32 years ago

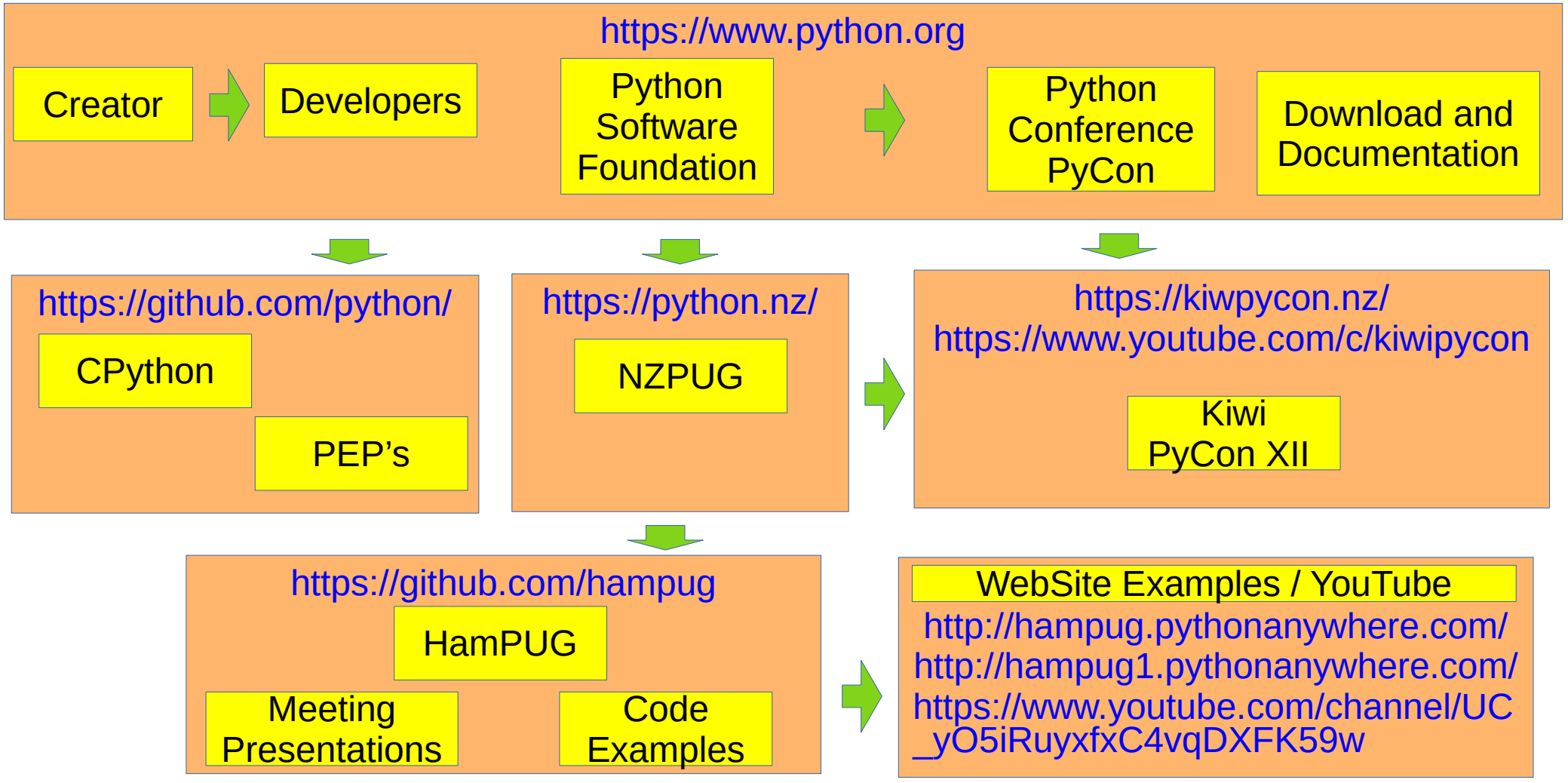
CPython:

- Reference implementation of the Python programming language.
- Written in C and Python.
- Default and most widely used implementation of the Python
- Defined as both an interpreter and a compiler. It compiles Python code into bytecode before interpreting it.

Licensing:

 https://www.python.org/

Python Sites Summary



External Sites for Example Python code

Stack Overflow

https://stackoverflow.com/questions/tagged/python/

You Tube videos

https://www.youtube.com/results?search_query=python

Program Creek

https://www.programcreek.com/python/

ChatGPT

https://chat.openai.com/

More?

Demo

Demo all links on previous two slides.

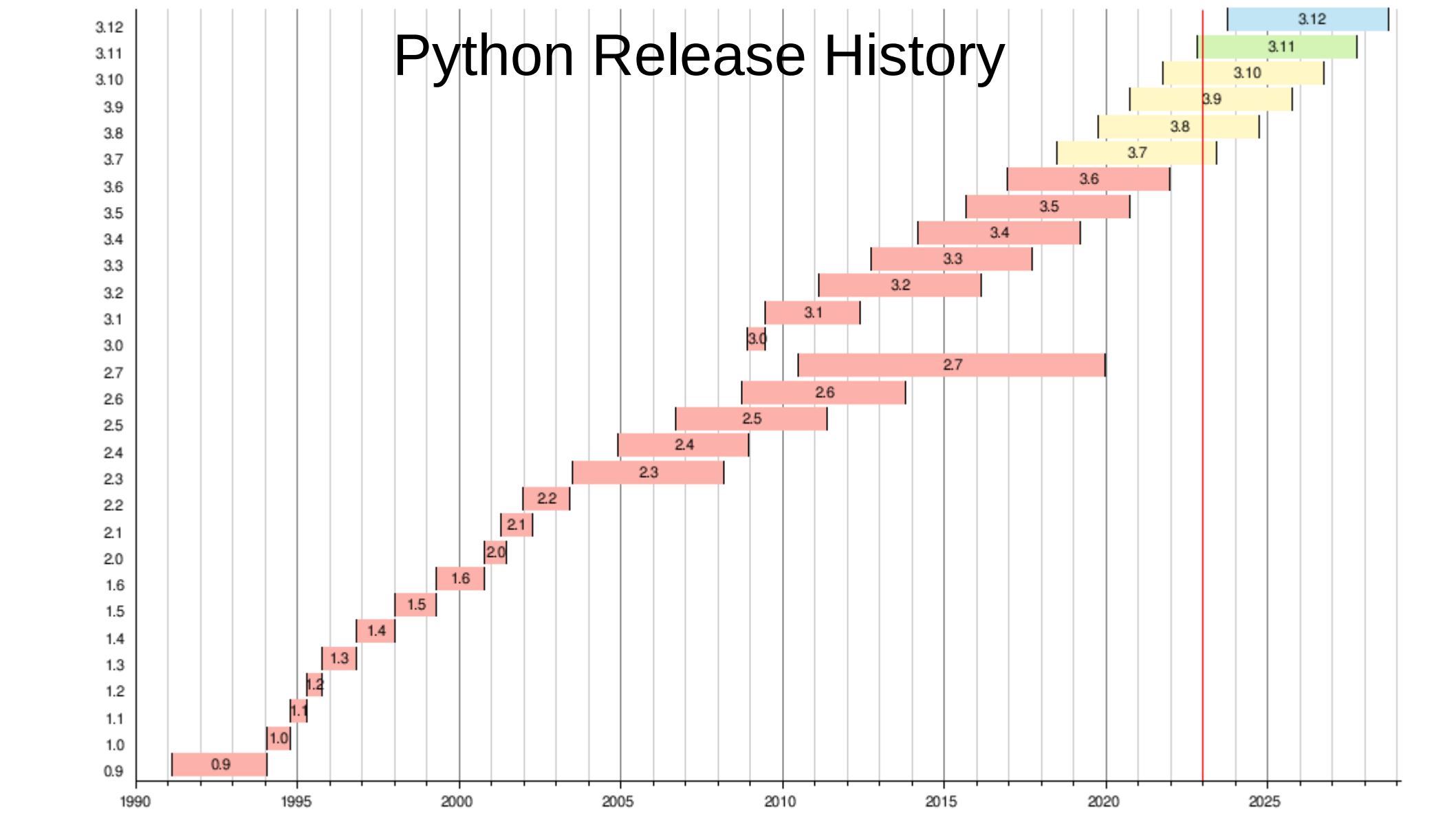
CPython Alternatives

Alternatives:

- Jython, written in Java for the Java virtual machine (JVM).
- PyPy, written in RPython and translated into C.
- IronPython, which is written in C#.
- Stackless Python
- MicroPython, written in C.
- CircuitPython, fork of MicroPython with emphasis on education.

Dialects:

- Cython. Superset of Python.
- Rpython. Restricted subset of Python
- Starlark. Language similar to Python.


Python Release History
3.12
3.11
3.10
3.9
3.8
3.7
3.6
3.5
3.4
3.3
3.2
3.1
3.0
2.7
2.6
2.5
2.4
2.3
2.2
2.1
2.0
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
1990
1995
2000
2005
2010
2015
2020
2025

Version	Latest micro version	Release date	End of full support	End of security fixes
0.9	0.9.9[2]	1991-02-20[2]	1993-07-29[a][2]	
1.0	1.0.4[2]	1994-01-26[2]	1994-02-15[a][2]	
1.1	1.1.1[2]	1994-10-11[2]	1994-11-10[a][2]	
1.2		1995-04-13[2]	Unsupported	
1.3		1995-10-13[2]	Unsupported	
1.4		1996-10-25[2]	Unsupported	
1.5	1.5.2[42]	1998-01-03[2]	1999-04-13[a][2]	
1.6	1.6.1[42]	2000-09-05[43]	2000-09[a][42]	
2.0	2.0.1[44]	2000-10-16[45]	2001-06-22[a][44]	
2.1	2.1.3[44]	2001-04-15[46]	2002-04-09[a][44]	
2.2	2.2.3[44]	2001-12-21[47]	2003-05-30[a][44]	
2.3	2.3.7[44]	2003-06-29[48]	2008-03-11[a][44]	
2.4	2.4.6[44]	2004-11-30[49]	2008-12-19[a][44]	
2.5	2.5.6[44]	2006-09-19[50]	2011-05-26[a][44]	
2.6	2.6.9[27]	2008-10-01[27]	2010-08-24[b][27]	2013-10-29[27]
2.7	2.7.18[32]	2010-07-03[32]	2020-01-01[c][32]	
3.0	3.0.1[44]	2008-12-03[27]	2009-06-27[51]	

Python Release History

3.1	3.1.5[52]	2009-06-27[52]	2011-06-12[53]	2012-04-06[52]
3.2	3.2.6[54]	2011-02-20[54]	2013-05-13[b][54]	2016-02-20[54]
3.3	3.3.7[55]	2012-09-29[55]	2014-03-08[b][55]	2017-09-29[55]
3.4	3.4.10[56]	2014-03-16[56]	2017-08-09[57]	2019-03-18[a][56]
3.5	3.5.10[58]	2015-09-13[58]	2017-08-08[59]	2020-09-30[58]
3.6	3.6.15[60]	2016-12-23[60]	2018-12-24[b][60]	2021-12-23[60]
3.7	*3.7.16*[61]	2018-06-27[61]	2020-06-27[b][61]	2023-06-27[61]
3.8	*3.8.16*[62]	2019-10-14[62]	2021-05-03[b][62]	2024-10[62]
3.9	*3.9.16*[63]	2020-10-05[63]	2022-05-17[b][63]	2025-10[63][64]
3.10[*needs update*]	*3.10.10*[65] [*needs update*]	2021-10-04[65]	2023-05[65]	2026-10[65]
3.11	3.11.2[66][*needs update*]	2022-10-24[66]	2024-05[66]	2027-10[66]
3.12	3.12.0[67]	2023-10-02[67]	2025-05[67]	2028-10[67]
Legend:	Old version	Older version, still maintained	**Latest version**	Latest preview version
	Future release			

https://en.wikipedia.org/wiki/History_of_Python

Bytecode

Python Compile to Bytecode

- Python compiles Python code into bytecode before interpreting it.
- Bytecode instructions may change with Python version.
- Local Imported modules once converted to bytecode are stored in __pycache__ sub-folder as xxx.cpython-3nn.pyc files.
- $ python -m compileall hello.py.

  Creates: __pycache__/hello.cpython-310.pyc
- Bytecode is interpreted by the “Python Virtual Machine”

- The implementation of the bytecode interpreter is in the file Python/ceval.c.

Display Op Names / Codes

```python
# Display all op codes and op names
import sys, dis

version = sys.version.split(" ")[0]
#print(version)
print("\nOpCodes for Python {} ByteCode".format(version))
op_name_dict = {}
for i in range(0,256): # max op code is 255.
    op_name = dis.opname[i]
    if op_name.startswith("<"):  # Unassigned opcodes display <255> etc.
        continue
    else:
        # Print op_code dec, op code hex, and op name
        print( "{:>3} {:>4}     {}".format(i, hex(i), op_name))
        op_name_dict[op_name] = i

# Sort op_name and display op name, op code dec and op code hex
print("\nOpNames for Python {} ByteCode".format(version))
for key in sorted(op_name_dict):
    hex_value = hex(op_name_dict[key])
    hex_value = (hex_value[2:])
    if len(hex_value) < 2:
        hex_value = "0" + hex_value
    print("{} {} {}".format(key, op_name_dict[key], hex_value))
```

Python 3.10 ByteCode. 127 Op Code Instructions.

OP Name	Dec	Hex	Op Name	Dec	Hex	Op Name	Dec	Hex
BEFORE_ASYNC_WITH	52	34	BUILD_STRING	157	9d	EXTENDED_ARG	144	90
BINARY_ADD	23	17	BUILD_TUPLE	102	66	FORMAT_VALUE	155	9b
BINARY_AND	64	40	CALL_FUNCTION	131	83	FOR_ITER	93	5d
BINARY_FLOOR_DIVIDE	26	1a	CALL_FUNCTION_EX	142	8e	GEN_START	129	81
BINARY_LSHIFT	62	3e	CALL_FUNCTION_KW	141	8d	GET_AITER	50	32
BINARY_MATRIX_MULTIPLY	16	10	CALL_METHOD	161	a1	GET_ANEXT	51	33
BINARY_MODULO	22	16	COMPARE_OP	107	6b	GET_AWAITABLE	73	49
BINARY_MULTIPLY	20	14	CONTAINS_OP	118	76	GET_ITER	68	44
BINARY_OR	66	42	COPY_DICT_WITHOUT_KEYS	34	22	GET_LEN	30	1e
BINARY_POWER	19	13	DELETE_ATTR	96	60	GET_YIELD_FROM_ITER	69	45
BINARY_RSHIFT	63	3f	DELETE_DEREF	138	8a	IMPORT_FROM	109	6d
BINARY_SUBSCR	25	19	DELETE_FAST	126	7e	IMPORT_NAME	108	6c
BINARY_SUBTRACT	24	18	DELETE_GLOBAL	98	62	IMPORT_STAR	84	54
BINARY_TRUE_DIVIDE	27	1b	DELETE_NAME	91	5b	INPLACE_ADD	55	37
BINARY_XOR	65	41	DELETE_SUBSCR	61	3d	INPLACE_AND	77	4d
BUILD_CONST_KEY_MAP	156	9c	DICT_MERGE	164	a4	INPLACE_FLOOR_DIVIDE	28	1c
BUILD_LIST	103	67	DICT_UPDATE	165	a5	INPLACE_LSHIFT	75	4b
BUILD_MAP	105	69	DUP_TOP	4	04	INPLACE_MATRIX_MULTIPLY	17	11
BUILD_SET	104	68	DUP_TOP_TWO	5	05	INPLACE_MODULO	59	3b
BUILD_SLICE	133	85	END_ASYNC_FOR	54	36	INPLACE_MULTIPLY	57	39

OP Code	Dec	Hex	Op Code	Dec	Hex	Op Code	Dec	Hex
INPLACE_OR	79	4f	LOAD_GLOBAL	116	74	SETUP_ANNOTATIONS	85	55
INPLACE_POWER	67	43	LOAD_METHOD	160	a0	SETUP_ASYNC_WITH	154	9a
INPLACE_RSHIFT	76	4c	LOAD_NAME	101	65	SETUP_FINALLY	122	7a
INPLACE_SUBTRACT	56	38	MAKE_FUNCTION	132	84	SETUP_WITH	143	8f
INPLACE_TRUE_DIVIDE	29	1d	MAP_ADD	147	93	SET_ADD	146	92
INPLACE_XOR	78	4e	MATCH_CLASS	152	98	SET_UPDATE	163	a3
IS_OP	117	75	MATCH_KEYS	33	21	STORE_ATTR	95	5f
JUMP_ABSOLUTE	113	71	MATCH_MAPPING	31	1f	STORE_DEREF	137	89
JUMP_FORWARD	110	6e	MATCH_SEQUENCE	32	20	STORE_FAST	125	7d
JUMP_IF_FALSE_OR_POP	111	6f	NOP	9	09	STORE_GLOBAL	97	61
JUMP_IF_NOT_EXC_MATCH	121	79	POP_BLOCK	87	57	STORE_NAME	90	5a
JUMP_IF_TRUE_OR_POP	112	70	POP_EXCEPT	89	59	STORE_SUBSCR	60	3c
LIST_APPEND	145	91	POP_JUMP_IF_FALSE	114	72	UNARY_INVERT	15	0f
LIST_EXTEND	162	a2	POP_JUMP_IF_TRUE	115	73	UNARY_NEGATIVE	11	0b
LIST_TO_TUPLE	82	52	POP_TOP	1	01	UNARY_NOT	12	0c
LOAD_ASSERTION_ERROR	74	4a	PRINT_EXPR	70	46	UNARY_POSITIVE	10	0a
LOAD_ATTR	106	6a	RAISE_VARARGS	130	82	UNPACK_EX	94	5e
LOAD_BUILD_CLASS	71	47	RERAISE	119	77	UNPACK_SEQUENCE	92	5c
LOAD_CLASSDEREF	148	94	RETURN_VALUE	83	53	WITH_EXCEPT_START	49	31
LOAD_CLOSURE	135	87	ROT_FOUR	6	06	YIELD_FROM	72	48
LOAD_CONST	100	64	ROT_N	99	63	YIELD_VALUE	86	56
LOAD_DEREF	136	88	ROT_THREE	3	03			
LOAD_FAST	124	7c	ROT_TWO	2	02			

Op Name Changes Example

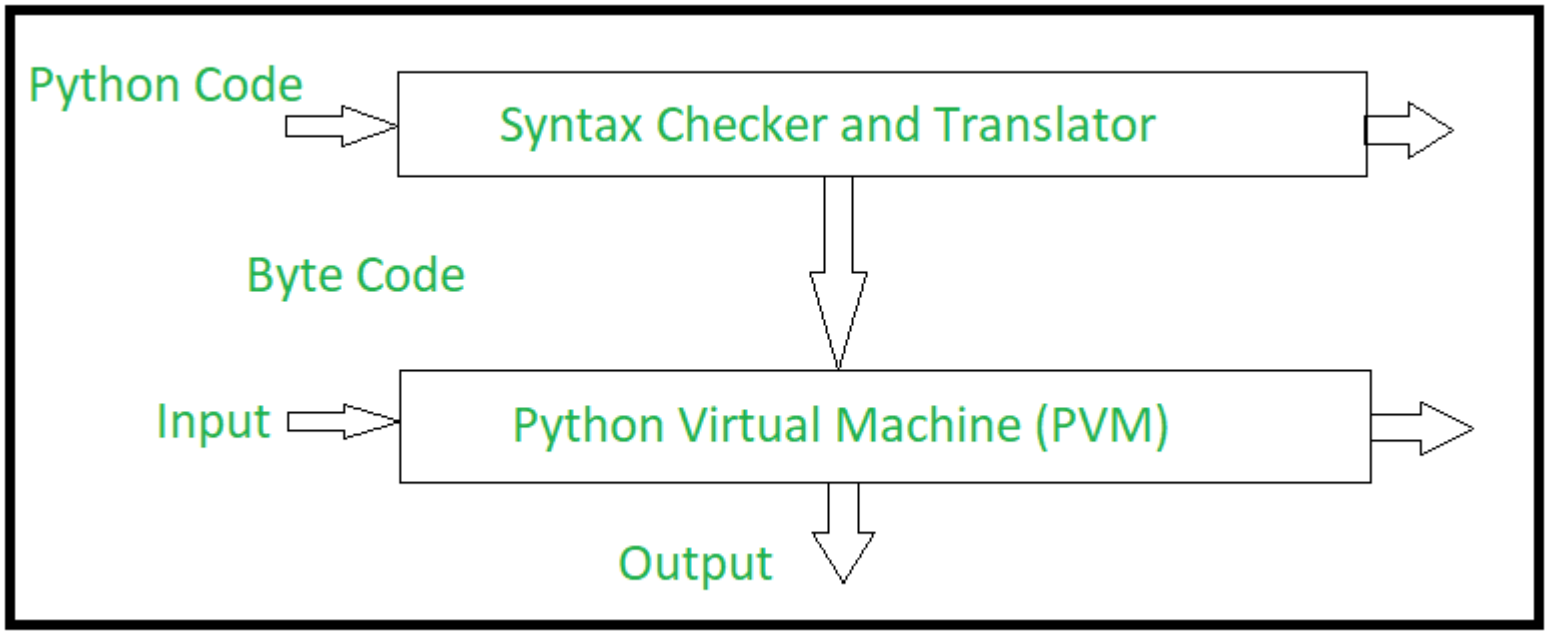
Python 3.10
Total Op Names: 127
Op Names in 3.10, but not in 3.11: 46

Python 3.11
Total Op Names: 110
Op Names in 3.11, but not in 3.10: 29

Op Names in both 3.10 and 3.11: 81

- https://leanpub.com/insidethepythonvirtualmachine/read
- https://nedbatchelder.com/blog/200804/the_structure_of_pyc_files.html
- https://towardsdatascience.com/understanding-python-bytecode-e7edaae8734d

Python Compile and Interpret - Simplified



The Python source code goes through the following to generate an executable code :

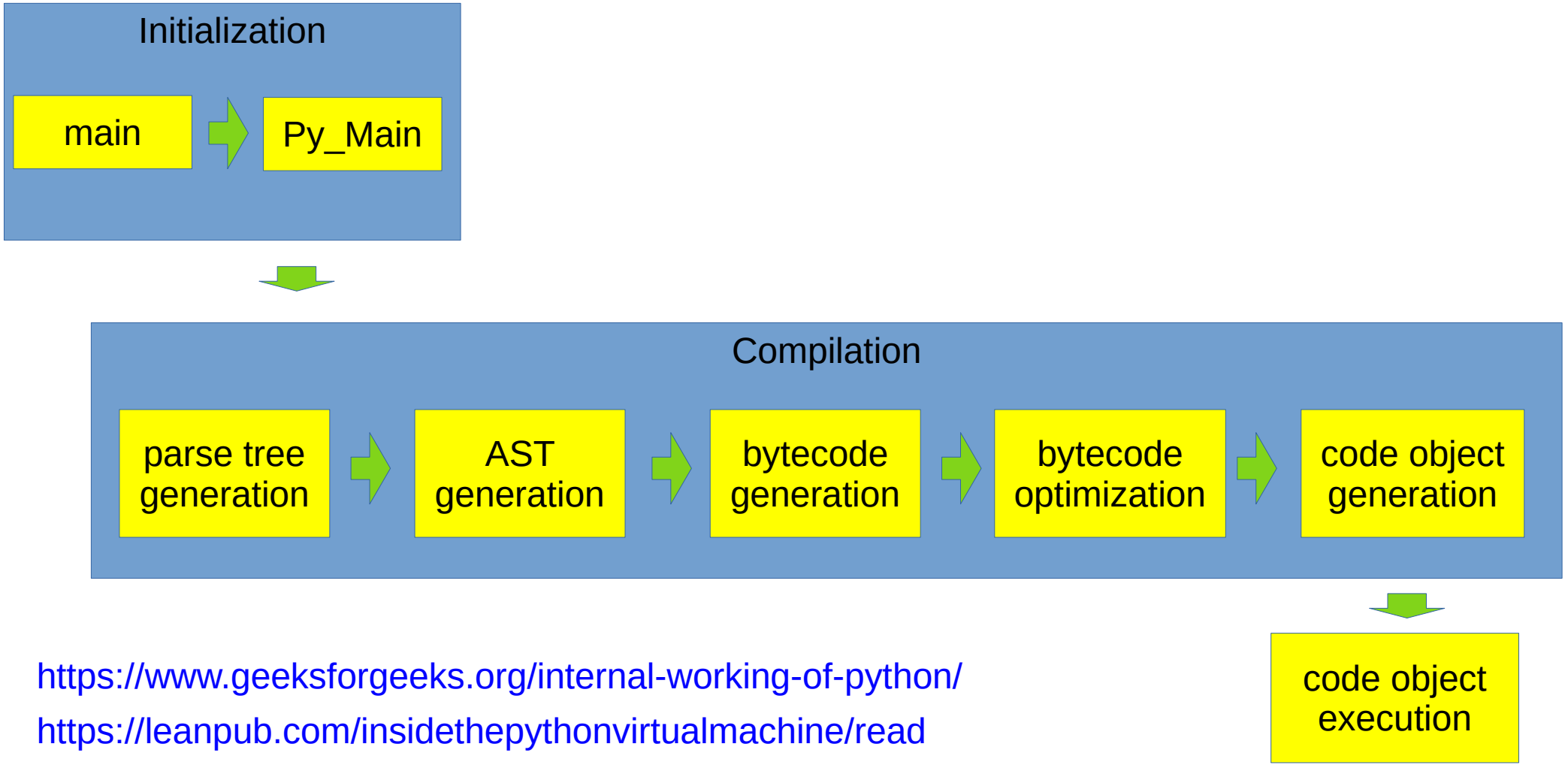
Step 1: The python compiler reads a python source code or instruction. Then it verifies that the instruction is well-formatted, i.e. it checks the syntax of each line. If it encounters an error, it immediately halts the translation and shows an error message.

Step 2: If there is no error, i.e. if the python instruction or source code is well-formatted then the compiler translates it into its equivalent form in an intermediate language called “Byte code”.

Step 3: Byte code is then sent to the Python Virtual Machine(PVM) which is the python interpreter. PVM converts the python byte code into machine-executable code. If an error occurs during this interpretation then the conversion is halted with an error message

https://www.geeksforgeeks.org/internal-working-of-python/

Flow during the execution of source code



https://www.geeksforgeeks.org/internal-working-of-python/

https://leanpub.com/insidethepythonvirtualmachine/read

Bytecode

```
1 print("hello world")
```

```
ian@hp:~/hampug/code$ python hello.py
hello world
```

```
ian@hp:~/hampug/code$ python -m compileall hello.py
Compiling 'hello.py'...
```

```
ian@hp:~/hampug/code$ ls -l __pycache__/hello.cpython-310.pyc
-rw-rw-r-- 1 ian ian 126 Apr  5 12:07 __pycache__/hello.cpython-310.pyc
```

```
ian@hp:~/hampug/code$ hexdump -C __pycache__/hello.cpython-310.pyc
00000000  6f 0d 0d 0a 00 00 00 00  c2 ba 2c 64 15 00 00 00  |o.........,d....|
00000010  e3 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  00 02 00 00 00 40 00 00  00 73 0c 00 00 00 65 00  |.....@...s....e.|
00000030  64 00 83 01 01 00 64 01  53 00 29 02 7a 0b 68 65  |d.....d.S.).z.he|
00000040  6c 6c 6f 20 77 6f 72 6c  64 4e 29 01 da 05 70 72  |llo worldN)...pr|
00000050  69 6e 74 a9 00 72 02 00  00 00 72 02 00 00 00 fa  |int..r....r.....|
00000060  08 68 65 6c 6c 6f 2e 70  79 da 08 3c 6d 6f 64 75  |.hello.py..<modu|
00000070  6c 65 3e 01 00 00 00 73  02 00 00 00 0c 00        |le>....s......|
0000007e
```

Disassemble .pyc file 1/2

```python
# view_pyc_file_3.7_up.py
# This will produce a disassembly of a .pyc file for python 3.7 and above.
import platform, time, sys, binascii, marshal, dis, struct

if sys.version_info.major == 3 and sys.version_info.minor < 7:
    sys.exit("For Python version 3.7 and above. Exiting...")


def view_pyc_file(path):
    """Read and display a content of the Python`s bytecode in a .pyc file."""
    with open(path, 'rb') as pyc_file:
        magic = pyc_file.read(4)
        bit_field = None
        timestamp = None
        hashstr = None
        size = None

        bit_field = int.from_bytes(pyc_file.read(4), byteorder=sys.byteorder)
        if 1 & bit_field == 1:
            hashstr = pyc_file.read(8)
        else:
            timestamp = pyc_file.read(4)
            size = pyc_file.read(4)
            size = struct.unpack('I', size)[0]

        code = marshal.load(pyc_file)

```

Program to Disassemble .pyc file 2/2

```
28      magic = binascii.hexlify(magic).decode('utf-8')
29      timestamp = time.asctime(time.localtime(struct.unpack('I', timestamp)[0]))
30
31      dis.disassemble(code)
32
33      print('-' * 80)
34      print(
35          'Python version: {}\nMagic code: {}\nTimestamp: {}\nSize: {}\nHash: {}\nBitfield: {}'
36          .format(platform.python_version(), magic, timestamp, size, hashstr, bit_field)
37      )
38
39
40  if __name__ == '__main__':
41      view_pyc_file(sys.argv[1])
```

view_pyc_file.py

```
ian@hp:~/hampug/code$ python view_pyc_file.py __pycache__/hello.cpython-310.pyc
  1           0 LOAD_NAME                0 (print)
              2 LOAD_CONST               0 ('hello world')
              4 CALL_FUNCTION            1
              6 POP_TOP
              8 LOAD_CONST               1 (None)
             10 RETURN_VALUE
------------------------------------------------------------------------------
Python version: 3.10.6
Magic code: 6f0d0d0a
Timestamp: Wed Apr  5 12:03:14 2023
Size: 21
Hash: None
Bitfield: 0
```

Op Codes in .pyc file

```
 0 LOAD_NAME                0 (print)
 2 LOAD_CONST               0 ('hello world')
 4 CALL_FUNCTION            1
 6 POP_TOP
 8 LOAD_CONST               1 (None)
10 RETURN_VALUE
```

Op Name	Decimal	Hex
LOAD_NAME	101	65
LOAD_CONST	100	64
CALL_FUNCTION	131	83
POP_TOP	1	01
LOAD_CONST	100	64
RETURN_VALUE	83	53

```
hexdump -C __pycache__/hello.cpython-310.pyc
00000000  6f 0d 0d 0a 00 00 00 00  c2 ba 2c 64 15 00 00 00  |o.........,d....|
00000010  e3 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  00 02 00 00 00 40 00 00  00 73 0c 00 00 00 65 00  |.....@...s....e.|
00000030  64 00 83 01 01 00 64 01  53 00 29 02 7a 0b 68 65  |d.....d.S.).z.he|
00000040  6c 6c 6f 20 77 6f 72 6c  64 4e 29 01 da 05 70 72  |llo worldN)...pr|
00000050  69 6e 74 a9 00 72 02 00  00 00 72 02 00 00 00 fa  |int..r....r.....|
00000060  08 68 65 6c 6c 6f 2e 70  79 da 08 3c 6d 6f 64 75  |.hello.py..<modu|
00000070  6c 65 3e 01 00 00 00 73  02 00 00 00 0c 00        |le>....s......|
0000007e
```

Additional data in .pyc file

```
Date
$ hex=642cbac2
$ echo "ibase=16; ${hex^^}" | bc
1680652994
$ date --date='@1680652994'
Wed 05 Apr 2023 12:03:14 NZST
```

```
Size
$ hex=00000015
$ echo "ibase=16; ${hex^^}" | bc
21
```

```
Magic code: 6f0d0d0a
Bit Field: 00000000
```

```
hexdump -C __pycache__/hello.cpython-310.pyc
00000000  6f 0d 0d 0a 00 00 00 00  c2 ba 2c 64 15 00 00 00  |o.........,d....|
00000010  e3 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  00 02 00 00 00 40 00 00  00 73 0c 00 00 00 65 00  |.....@...s....e.|
00000030  64 00 83 01 01 00 64 01  53 00 29 02 7a 0b 68 65  |d.....d.S.).z.he|
00000040  6c 6c 6f 20 77 6f 72 6c  64 4e 29 01 da 05 70 72  |llo worldN)...pr|
00000050  69 6e 74 a9 00 72 02 00  00 00 72 02 00 00 00 fa  |int..r....r.....|
00000060  08 68 65 6c 6c 6f 2e 70  79 da 08 3c 6d 6f 64 75  |.hello.py..<modu|
00000070  6c 65 3e 01 00 00 00 73  02 00 00 00 0c 00        |le>....s......|
0000007e
```

Tools

Jupyter Notebook

https://jupyter.org/

Install the classic Jupyter Notebook with:

```
$ pip install notebook
```

OR:

```
$ sudo apt install jupyter-notebook
```

To run the notebook:

```
$ jupyter-notebook
```

IDE’s

Geany

https://geany.org/

```
$ sudo apt install geany
```

PyCharm

https://www.jetbrains.com/pycharm/

Download:

https://www.jetbrains.com/pycharm/download/#section=linux

Git / Github

Git

https://en.wikipedia.org/wiki/Git

Install:

```
$ sudo apt install git
```

Github

https://github.com

Python mailing-list

A general discussion list for the Python programming language.

Subscribe at:

https://mail.python.org/mailman/listinfo/python-list

NZPUG Mailing list:

https://groups.google.com/g/nzpug

Taylor Expansion Series – Raspberry Pi Pico ADC

Taylor Expansion Series – Raspberry Pi Pico ADC

Pico module has 5 x 12 bit ADC.

Inputs in the range 0 to 3.3V will output 0 to 4095.

Micropython converts this 12 bit value to a 16 bit value:

0 to 65535 in steps of “mostly” 16, but sometimes 17.

https://en.wikipedia.org/wiki/Taylor_series

Taylor Expansion Series – Raspberry Pi Pico ADC

```
35  STATIC uint16_t adc_config_and_read_u16(uint32_t channel) {
36      adc_select_input(channel);
37      uint32_t raw = adc_read();
38      const uint32_t bits = 12;
39      // Scale raw reading to 16 bit value using a Taylor expansion (for 8 <= bits <= 16)
40      return raw << (16 - bits) | raw >> (2 * bits - 16);
41  }
```

https://github.com/micropython/micropython/blob/master/ports/rp2/machine_adc.c

```
4  def taylor_series(raw_value):
5      bits = 12
6      return raw_value << (16 - bits) | raw_value >> (2 * bits - 16)
7
8  taylor_value = 0
9  for i in range(4097):
10     taylor_previous = taylor_value
11     taylor_value = taylor_series(i)
12     print(i, taylor_value, taylor_value-taylor_previous )
```

Taylor

```
0 0 0
1 16 16
2 32 16
3 48 16
4 64 16
5 80 16
6 96 16
7 112 16
8 128 16
9 144 16
10 160 16
11 176 16
12 192 16
13 208 16
14 224 16
15 240 16
16 256 16
17 272 16
18 288 16
19 304 16
```

```
247 3952 16
248 3968 16
249 3984 16
250 4000 16
251 4016 16
252 4032 16
253 4048 16
254 4064 16
255 4080 16
256 4097 17
257 4113 16
258 4129 16
259 4145 16
260 4161 16
261 4177 16
262 4193 16
263 4209 16
264 4225 16
265 4241 16
266 4257 16
```

```
4077 65247 16
4078 65263 16
4079 65279 16
4080 65295 16
4081 65311 16
4082 65327 16
4083 65343 16
4084 65359 16
4085 65375 16
4086 65391 16
4087 65407 16
4088 65423 16
4089 65439 16
4090 65455 16
4091 65471 16
4092 65487 16
4093 65503 16
4094 65519 16
4095 65535 16
4096 65552 17
```

```
# Increment = 16, except...
0 0 0
1 16 16

256 4097 17
512 8194 17
768 12291 17
1024 16388 17
1280 20485 17
1536 24582 17
1792 28679 17
2048 32776 17
2304 36873 17
2560 40970 17
2816 45067 17
3072 49164 17
3328 53261 17
3584 57358 17
3840 61455 17

4095 65535 16

4096 65552 17
```

Taylor – 16 bit to 12 bit

```
18      list_16_bit.append(taylor_value)
19  #print(list_16_bit)
20
21  for bit_16 in list_16_bit:
22      bit_12 = bit_16 >> 4
23      print(bit_16, bit_12)
```

```
0 0
16 1
32 2
48 3
64 4
80 5
96 6
112 7
128 8
144 9
160 10
176 11
```

```
4016 251
4032 252
4048 253
4064 254
4080 255
4097 256
4113 257
4129 258
4145 259
4161 260
4177 261
```

```
65375 4085
65391 4086
65407 4087
65423 4088
65439 4089
65455 4090
65471 4091
65487 4092
65503 4093
65519 4094
65535 4095
```

Demo

Demo Pico module on breadboard with 100K ohm trim pot between +3.3V and ground.

Feed the trim pot wiper into the ADC channel.

Run adc_1.py to sample ADC every second...

While turning the trim pot through full range.

Output of program should be from 0 to 4095 (i.e. 12 bit range).

...end