Python Under the Hood

Understand the inner workings of Python

By Arianne Dee

Table of Contents

Survey (single choice)

- What is your level of Python experience?
  - Total beginner
  - Beginner
  - Advanced beginner
  - Intermediate
  - Advanced intermediate
  - Expert

Survey (multi-choice)

- What do you use Python for?
  - Data science / data analysis
  - Scientific computing / machine learning
  - Server automation
  - Scripting
  - Web applications
  - Command line applications
  - Desktop applications
  - Internet of things
  - Teaching

Survey (multi-choice)

- What are you most interested to cover?
  - CPython intro
  - Compiler/interpreter
  - Namespaces/scope
  - Objects, ids, and collections
  - IDE features and debuggers
  - Data model overview
  - Dunder/special methods
  - Metaprogramming

About Me

- Instructor for O'Reilly:
  - 8 Live Trainings
  - 4 Videos
  - Interactive Lab: Python Foundations

- 10+ years of software development, mostly in Python and Django web development
  - games, startups, consulting, agencies, freelance, education








Course Goals

- Improve your Python programming/debugging skills
- Give you more context to understand Python at a deeper level
- Improve your ability to read source code
- Provide a jumping off point to do more research in a specific topic

You'll understand these great videos

- **Beyond PEP 8 -- Best practices for beautiful intelligible code** - video
  - *Raymond Hettinger*
- **Stupid things I've done with Python** - video
  - *Mark Smith*
- **Elegant Solutions For Everyday Python Problems** - video
  - *Nina Zakharenko*

Course Outline

- 0:00 – Intro and setup
- 0:10 – **Under the hood**
  - Break (15 min)
- 1:50 – **Everything is an object**
  - Break (15 min)
- 3:30 – **Useful dunders**
- 4:55 – Course wrap-up

Questions and Breaks

- Use attendee chat throughout class
  - Off-topic questions go in Q&A widget
- 2 x 15 min breaks
  - Step away or work through code
  - I'll answer questions in the Q&A widget
- Email more in-depth questions at arianne.dee.studios@gmail.com

Under the Hood

What is Python?

A programming language

- High-level
- Object-oriented
- Dynamically typed

Python

- The language is defined by the **Python Language Reference**
  - https://docs.python.org/3/reference/index.html
- A new version is released yearly (usually October)
  - https://devguide.python.org/versions/
- **P**ython **E**nhancement **P**roposals (PEPs) outline changes
  - PEPs can be rejected, accepted or planned for a future release

Useful PEPs

- **PEP 0** – Index of Python Enhancement Proposals (PEPs)
- **PEP 1** – PEP Purpose and Guidelines
- **PEP 8** – Style Guide for Python Code
  - Use 4 spaces for indentation, snake case for variables, etc.
- **PEP 20** – The Zen of Python
  - Beautiful is better than ugly
- **PEP 257** – Docstring Conventions
- **PEP 404** – Python 2.8 Un-release Schedule
  - Why you should stop using Python 2

PEPs – Python Enhancement Proposals

- **Accepted PEPs**
  - accepted; may not be implemented yet
- **Open PEPs**
  - under consideration
- **Finished PEPs**
  - done, with a stable interface
- **Abandoned, Withdrawn, and Rejected PEPs**

PEPs – Python Enhancement Proposals

- **Meta-PEPs**
  - PEPs about PEPs or Processes
  - E.g. PEP 8 - Style Guide for Python Code
- **Other Informational PEPs**
  - E.g. PEP 20 - The Zen of Python
- **Provisional PEPs**
  - provisionally accepted; interface may still change
- **Historical Meta-PEPs and Informational PEPs**

What did you download and run?

A Python interpreter

- The CPython reference implementation

CPython

- Reference implementation of the Python language
- Most compliant with specifications (documentation/accepted PEPs)
- Written in C
- C-bindings

Common Python Interpreter Options

- **CPython** - default

Other common alternatives:

- **PyPy** – Just-in-time compiler that supports 2.7 and 3.8
- **Stackless Python** – Fork of CPython with microthreads
- **MicroPython/CircuitPython** – optimized for microcontrollers
- **Cython** – compiles Python into C extensions
- **Jython** – Python for the JVM (Java) – supports 2.7
- **IronPython** – written in C#, can use .NET libraries – supports 2.7 and 3.4

Other Python Interpreter Options

- **GraalVM** – Just-in-time compiler for the JVM – supports 3.11
- **IntelPython** – High performant for data and AI – supports 3.9
- **RustPython** – for Rust apps or WebAssembly – supports 3.11

- … more? Mention in chat so I can add to the list

Adding C-extensions to Python

1. Install locally, in local folder, without editing CPython or the Python interpreter

2. Install in your system Python interpreter, without editing CPython

3. Edit the CPython source code and compile to a new Python executable

xyzmodule.c

- Building a C-extension for CPython

```
>>> import xyz
>>> xyz.echo('Hello, world’)
Hello, world
Hello, world
```

Adding C-extensions to Python

1. Install locally, in local folder, without editing CPython or the Python interpreter

   `$ pip install setuptools` (if running 3.12+)

   ```
   $ python setup.py build_ext –inplace
   ```

   - This will add a shared object file (.so) to your local directory.
   - If the .so file is on your Python path (see sys.path), you can import `xyz` as a module.
   - Quick and easy, but not recommended for larger projects

CPython

CPython

- **Source code**: https://github.com/python/cpython

- Guide to working with CPython:
  - https://realpython.com/cpython-source-code-guide/

- Python Developer's Guide:
  - https://devguide.python.org/
  - CPython source code:
    - https://devguide.python.org/internals/exploring/

CPython Source Code

```
cpython/
│
├── Doc       ← Source for the documentation
├── Grammar   ← The computer-readable language definition
├── Include   ← The C header files
├── Lib       ← Standard library modules written in Python
├── Mac       ← macOS support files
├── Misc      ← Miscellaneous files
├── Modules   ← Standard Library Modules written in C
├── Objects   ← Core types and the object model
├── Parser    ← The Python parser source code
├── PC        ← Windows build support files
├── PCbuild   ← Windows build support files for older Windows versions
├── Programs  ← Source code for the python executable and other binaries
├── Python    ← The CPython interpreter source code
└── Tools     ← Standalone tools useful for building or extending Python
```

Source Code Layout

For a Python module, the typical layout is:

- `Lib/<module>.py`
- `Modules/_<module>.c` (if there's also a C accelerator module)
- `Lib/test/test_<module>.py`
- `Doc/library/<module>.rst`

For an extension module, the typical layout is:

- `Modules/<module>module.c`
- `Lib/test/test_<module>.py`
- `Doc/library/<module>.rst`

For Built-in Types, the typical layout is:

- `Objects/<builtin>object.c`
- `Lib/test/test_<builtin>.py`
- `Doc/library/stdtypes.rst`

For Built-in Functions, the typical layout is:

- `Python/bltinmodule.c`
- `Lib/test/test_builtin.py`
- `Doc/library/functions.rst`

Some exceptions to these layouts are:

- built-in type `int` is at `Objects/longobject.c`
- built-in type `str` is at `Objects/unicodeobject.c`
- built-in module `sys` is at `Python/sysmodule.c`
- built-in module `marshal` is at `Python/marshal.c`
- Windows-only module `winreg` is at `PC/winreg.c`

https://devguide.python.org/internals/exploring/

Compiling CPython (Unix)

1. Install dependencies
2. Run the configure script
   - Generates a Makefile
   - Set options for features here
3. Build a CPython executable by running the Makefile
   - Generates a python executable

- Instructions for Mac/Linux/Windows

Resources - CPython

- [RealPython] **Your Guide to the CPython Source Code**
  - https://realpython.com/cpython-source-code-guide/

- [Python Developer's Guide] **Compiler design**
  - https://devguide.python.org/internals/compiler/

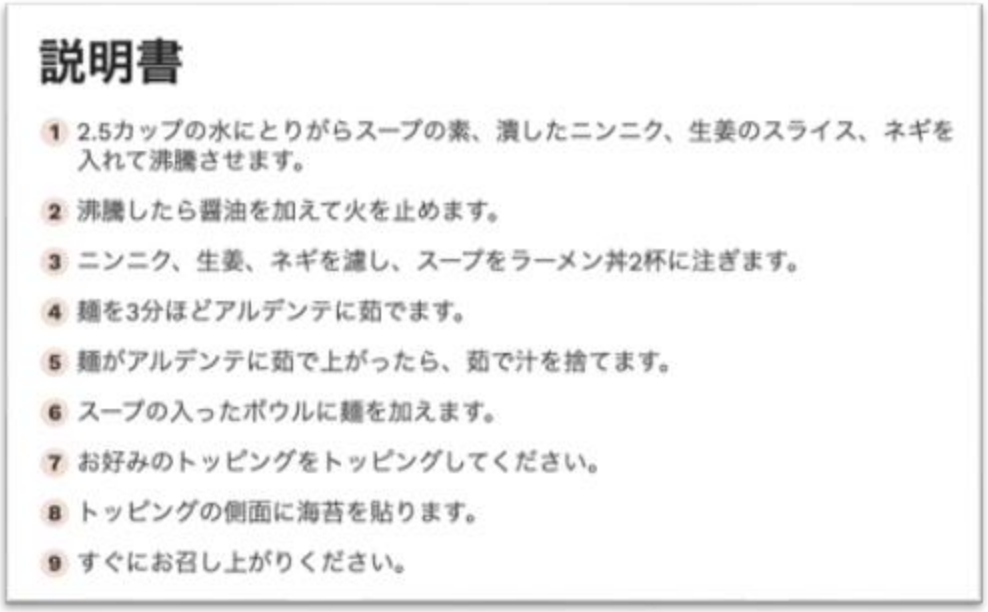
- [Python Developer's Guide] **The bytecode interpreter**
  - https://devguide.python.org/internals/interpreter/

Is it compiled or interpreted?

- Compiled
  - Translated into a lower-level language
- Interpreted
  - Code is run by an interpreter, not the target machine

Example

- You have a recipe for ramen written in Japanese

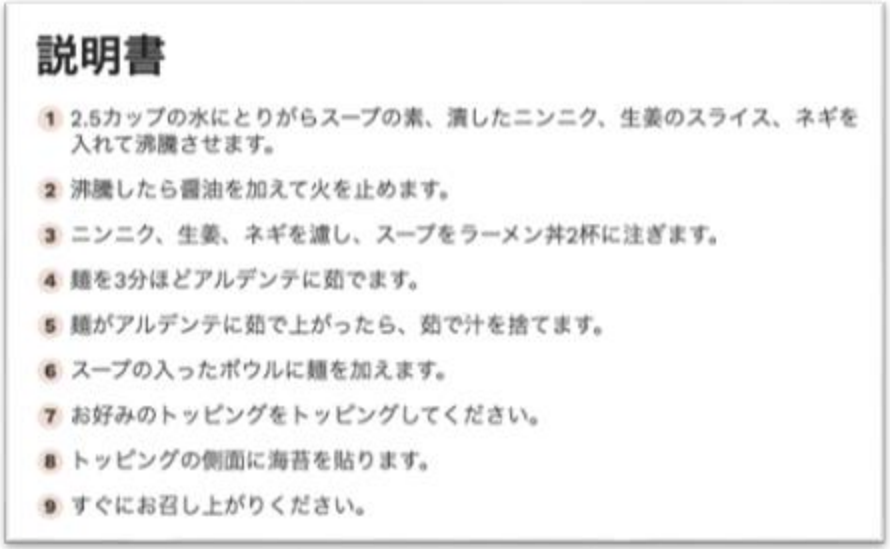


説明書

1. 2.5カップの水にとりがらスープの素、潰したニンニク、生姜のスライス、ネギを入れて沸騰させます。
2. 沸騰したら醤油を加えて火を止めます。
3. ニンニク、生姜、ネギを濾し、スープをラーメン丼2杯に注ぎます。
4. 麺を3分ほどアルデンテに茹でます。
5. 麺がアルデンテに茹で上がったら、茹で汁を捨てます。
6. スープの入ったボウルに麺を加えます。
7. お好みのトッピングをトッピングしてください。
8. トッピングの側面に海苔を貼ります。
9. すぐにお召し上がりください。

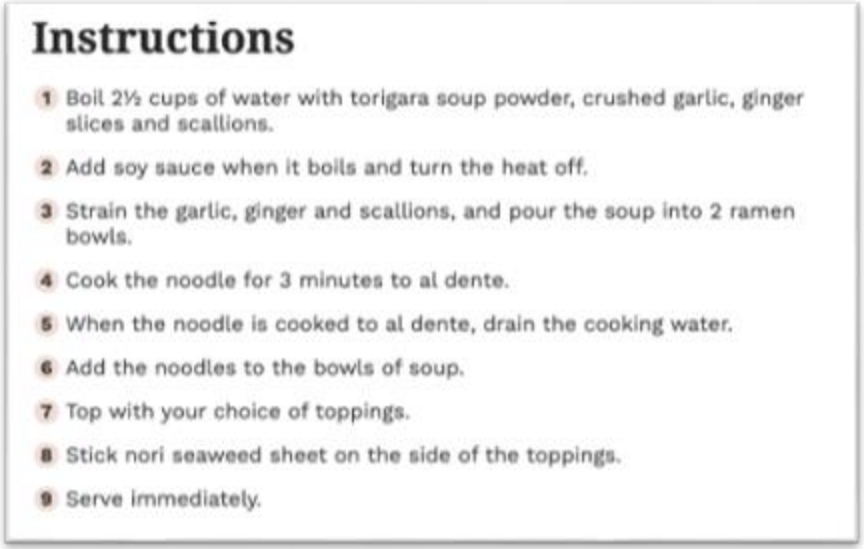
Compiled

- Code is translated from source language to another language



説明書

1. 2.5カップの水にとりがらスープの素、潰したニンニク、生姜のスライス、ネギを入れて沸騰させます。
2. 沸騰したら醤油を加えて火を止めます。
3. ニンニク、生姜、ネギを濾し、スープをラーメン丼2杯に注ぎます。
4. 麺を3分ほどアルデンテに茹でます。
5. 麺がアルデンテに茹で上がったら、茹で汁を捨てます。
6. スープの入ったボウルに麺を加えます。
7. お好みのトッピングをトッピングしてください。
8. トッピングの側面に海苔を貼ります。
9. すぐにお召し上がりください。



Instructions

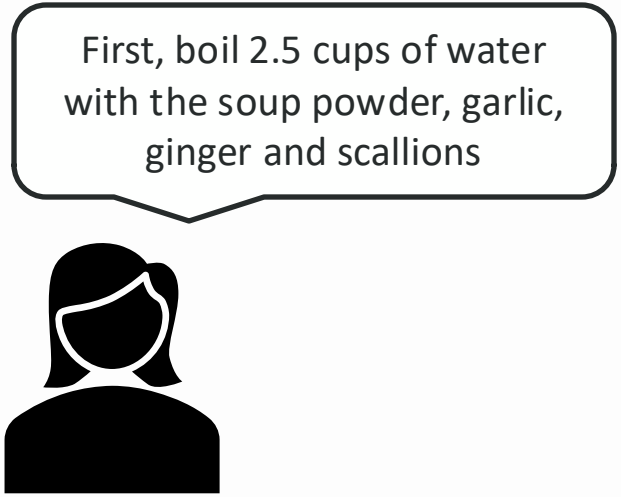
1. Boil 2½ cups of water with torigara soup powder, crushed garlic, ginger slices and scallions.
2. Add soy sauce when it boils and turn the heat off.
3. Strain the garlic, ginger and scallions, and pour the soup into 2 ramen bowls.
4. Cook the noodle for 3 minutes to al dente.
5. When the noodle is cooked to al dente, drain the cooking water.
6. Add the noodles to the bowls of soup.
7. Top with your choice of toppings.
8. Stick nori seaweed sheet on the side of the toppings.
9. Serve immediately.

Interpreted

- An interpreter is needed to translate each instruction, line by line

説明書

1. 2.5カップの水にとりがらスープの素、潰したニンニク、生姜のスライス、ネギを入れて沸騰させます。
2. 沸騰したら醤油を加えて火を止めます。
3. ニンニク、生姜、ネギを濾し、スープをラーメン丼2杯に注ぎます。
4. 麺を3分ほどアルデンテに茹でます。
5. 麺がアルデンテに茹で上がったら、茹で汁を捨てます。
6. スープの入ったボウルに麺を加えます。
7. お好みのトッピングをトッピングしてください。
8. トッピングの側面に海苔を貼ります。
9. すぐにお召し上がりください。



Is it compiled or interpreted?

- CPython is both!
- Compiled into bytecode
- Bytecode is interpreted by the interpreter

Compiled

- Code is translated from Python to bytecode

```
print(“Hello, world”)
```

```
2           0 LOAD_GLOBAL              0 (print)
            2 LOAD_CONST               1 ('Hello, World!')
            4 CALL_FUNCTION            1
```

Interpreted

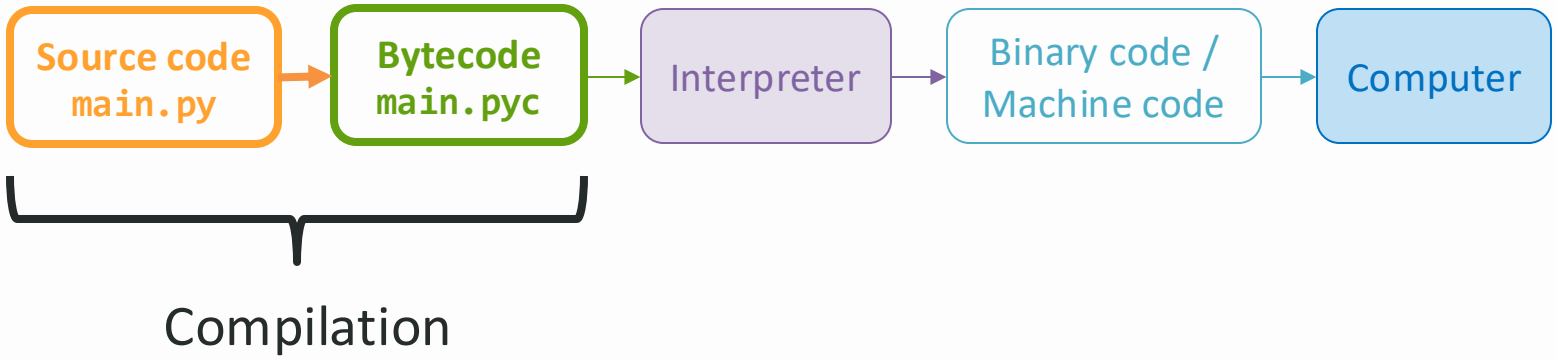
- The bytecode is run, line by line, by the **P**ython **v**irtual **m**achine
- The PVM (written in C) translates each bytecode instruction to machine code for the processor to run

```
2           0 LOAD_GLOBAL              0 (print)
            2 LOAD_CONST               1 ('Hello, World!')
            4 CALL_FUNCTION            1
```

```
00101111001001101100010010100...
```

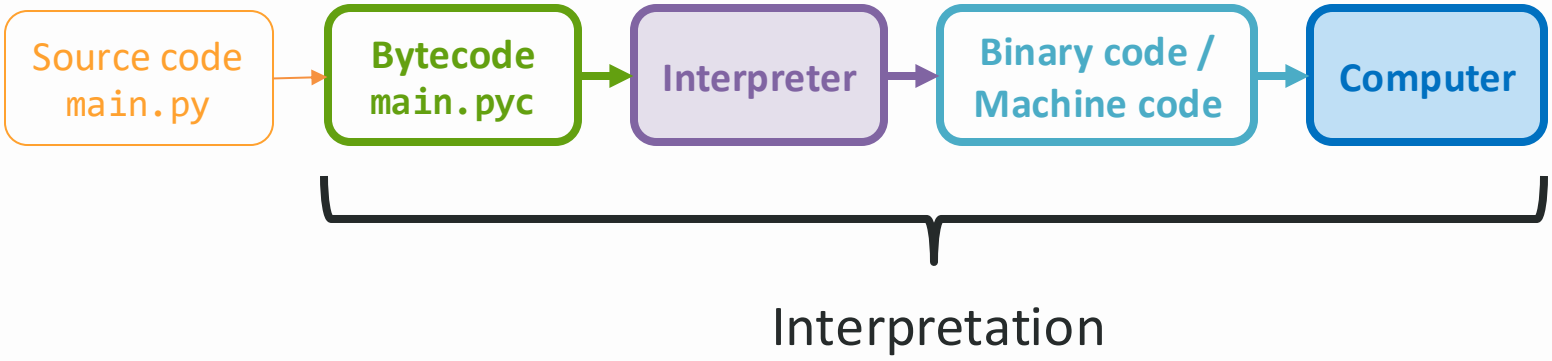
Lifecycle of a Python program

Lifecycle of a Python program



- Python compiles .py files to cached .pyc files
- .pyc files contain bytecode that only CPython understands

Lifecycle of a Python program



- The interpreter transforms the bytecode to machine code and executes it

Lifecycle of a Python program

- CPython compiles and interprets (runs) in one step
  - `$ python main.py`

- Non-interpreted languages have at least 2 steps
  - Compile code to machine code
  - Run machine code

Compilation

1. Tokenization
2. Parsing
3. AST tree construction
4. Byte code generation

Interpretation

- Bytecode loaded into Python runtime
- A **virtual machine** (written in C) interprets the bytecode, line-by-line, and transforms it into machine code
- Machine code is executed, line-by-line

Python errors

- Compilation errors -> **syntax errors**
  - e.g. SyntaxError, IndentationError and TabError

- Interpretation errors -> **runtime errors**
  - Occur when running a line of code that raises and exception
  - Get a traceback of the call stack when the exception was raised

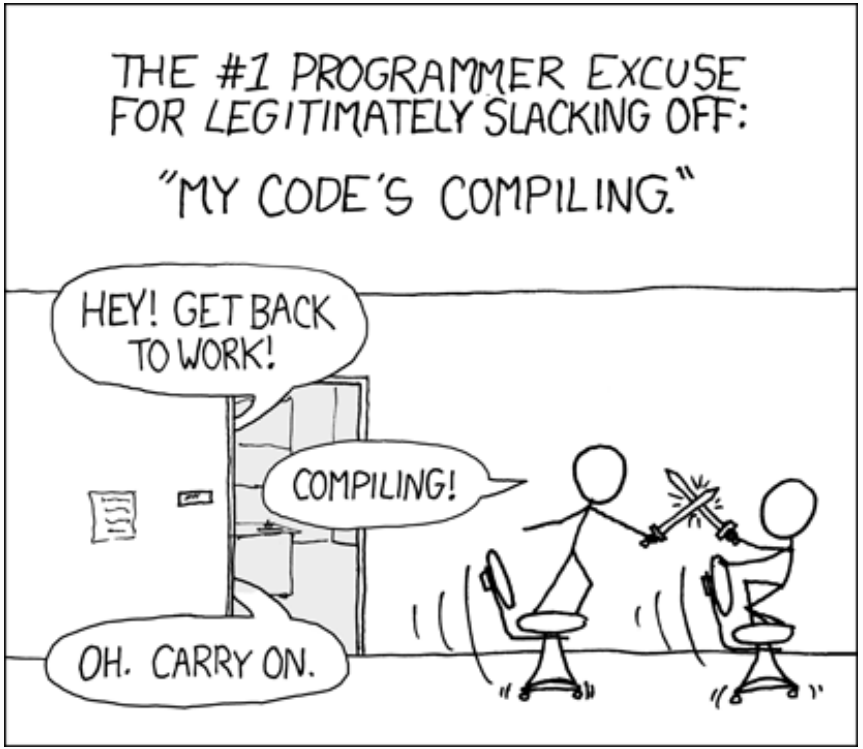
Cons

- Slower
  - Must translate each line when running
    - “Interpretive overhead”
  - Must access variables at runtime rather than compile time
- More errors can be caught with fully compiled languages
  - Don’t bother executing programs with critical problems
- Interpreter needed on target machine to run code

Pros

- No extra build-step
  - Faster to edit code and see changes
- Easier to debug
  - Syntax errors are easy to identify and fix
  - Runtime errors allow you to more easily see the source code and current state
- Can distribute source code

XKCD – 303 – Compiling



https://xkcd.com/303/

JIT Compilers

- **J**ust-**I**n-**T**ime (JIT) compilers analyze code during execution.
- Can pre-translates the most critical parts into chunks of machine code.
- Slower during analysis, but long running programs end up running faster overall.
- PyPy supports 2.7 and 3.8
- Numba supports CPython and Numpy
- Python 3.13 started work on an experimental JIT compiler

Experimental JIT

- Python 3.13 added an optional experimental JIT
- Need to compile Python from source code with `--enable-experimental-jit` option
- Uses copy-and-patch technique that pattern matches bytecode and patches in already translated machine code when needed
- Useful in long running programs with lots of repeated code (like in loops)

Resources – Interpreted vs Compiled

- [YouTube] **What is the Python Interpreter? (How does Python Work?)**
  - https://www.youtube.com/watch?v=BkHdmAhapws

- [freeCodeCamp] **Interpreted vs Compiled Programming Languages: What's the Difference?**
  - https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/

CPython details

Compilation

1. Tokenization
2. Parsing
3. AST tree construction
4. Byte code generation

CPython's Compilation Steps

1. **Tokenization** (lexical analysis)
   - Transforms string of characters into tokens
2. **Parsing** (syntax analysis)
   - Checks for correct syntax (tokens are in a valid order)
   - Produces an Abstract Syntax Tree (AST)
3. **AST tree construction**
   - Hierarchical representation of the code
   - Shows relationship between the tokens
4. **Byte code generation**

Note: There are actually more steps internally that aren't covered

Tokenization

- Python uses a **lexer/tokenizer** to turn a string of text into tokens

- In English, the phrase “I love Python!”:
  - **Tokenizer** recognizes the tokens [I, love, Python, !]
  - **Lexer** recognizes the tokens [I (pronoun), love (verb), Python (noun), ! (punctuation)]

Tokenization

- In Python, the code `print('Hello')`:
  - **Lexer** recognizes the tokens:
    - `print` (NAME)
    - `(` (LPAR)
    - `'Hello'` (STRING)
    - `)` (RPAR)

Tokenization

- **Token** - a meaningful unit of text
  - E.g. keyword, name, indent, symbol (like `(`,`)`,`:`,`.`,`+`,`-`)
- See `cpython/Grammar/Tokens`

```
ENDMARKER
NAME
NUMBER
STRING
NEWLINE
INDENT
DEDENT
LPAR                    '('
RPAR                    ')'
LSQB                    '['
RSQB                    ']'
COLON                   ':'
COMMA                   ','
```

Tokenization

- Test out Python's tokenizer
  - `python –m tokenize <file.py>` - docs

- On `print('Hello')`:

```
0,0-0,0:            ENCODING       'utf-8'
1,0-1,5:            NAME           'print'
1,5-1,6:            OP             '('
1,6-1,19:           STRING         "'Hello'"
1,19-1,20:          OP             ')'
1,20-1,21:          NEWLINE        '\n'
2,0-2,0:            ENDMARKER      ''
```

example_1_tokenize.py

Parsing

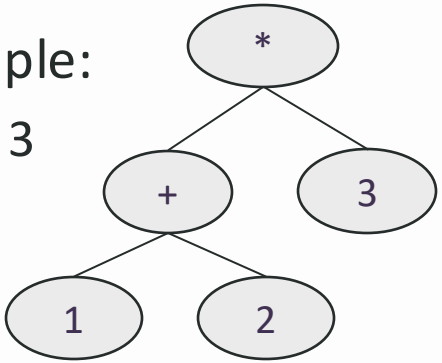
- Turns the tokens into an **Abstract Syntax Tree** (AST)
- Creates meaning from the tokens

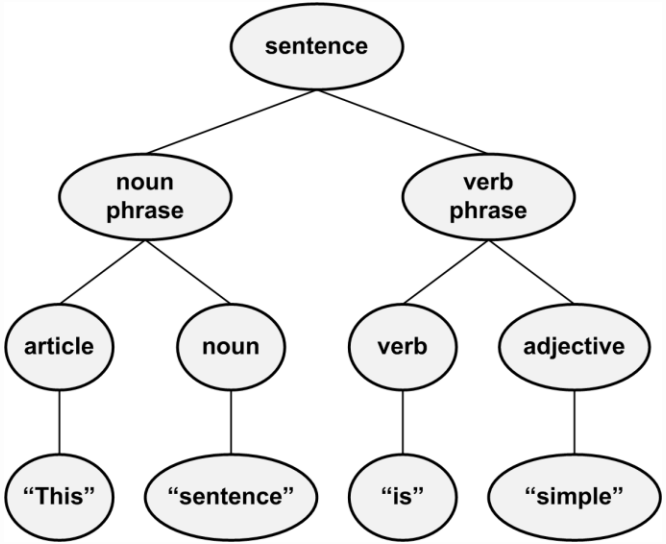
- Note:
  - 3.8- used an LL(1) parser
  - 3.9+ uses a PEG parser

- See `cpython/Grammar/python.gram` to see the grammar

Abstract Syntax Trees

- ASTs are an abstract representation of the source code
- English example:
  - “This sentence is simple”
- Math example:
  - (1 + 2) * 3



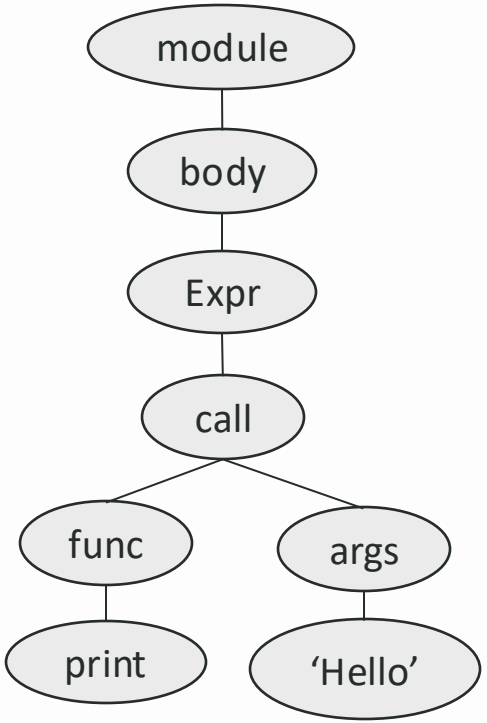


https://www.codeproject.com/Articles/5310967/Analyzing-Python-with-the-AST-Package

Abstract Syntax Trees

- ASTs are an abstract representation of the source code - docs
- AST for `print('Hello')`

```
Module(
    body=[
        Expr(
            value=Call(
                func=Name(id='print', ctx=Load()),
                args=[
                    Constant(value='Hello')],
                keywords=[]))],
    type_ignores=[])
```



example_2_parse.py

Compile to Bytecode

- AST gets turned into bytecode
- Low-level, platform independent representation of the code
- Saved as .pyc files
- Can be distributed and run by the Python Interpreter
  - `$ python my_compiled_file.pyc`

Compile to Bytecode

- `$ python -m py_compile <file.py>` - docs
  - Creates a .pyc file of binary bytecode

- E.g. `$ python -m py_compile hello.py`
  - Compiles `hello.py` to `__pycache__/hello.cpython-3xx.pyc`

- `$ python -m compileall <dir>` - docs
  - Compiles all files in a directory to .pyc files

Compile to Bytecode

- Binary files consist of a bunch of 0's and 1's
- .pyc will look like gibberish if you try to read them
- View the bytecode instructions with `example_3_view_pyc.py`

```
$ cat __pycache__/hello.cpython-312.pyc
�
|/�e<���d�Zed�y)c�
�td|z�y)NzHello )�print)�names
hello.py�greetrs��
�(�T�/���worldN)r�rr<module>r
s����g�r%
```

```
  0           0 RESUME                   0

  1           2 PUSH_NULL
              4 LOAD_NAME                0 (print)
              6 LOAD_CONST               0 ('Hello')
              8 CALL                     1
             16 POP_TOP
             18 RETURN_CONST             1 (None)
--------------------------------------------------
Python version: 3.12.1
Magic code: cb0d0d0a
Timestamp: Fri Jan 12 15:43:11 2024
Size: 14
Hash: None
Bitfield: 0
```

example_3_view_pyc.py

View Bytecode from a File

- To view the bytecode instructions directly from a .py file
  - `$ python -m dis <file.py>`

- For a file containing: `print('Hello')`

```
0           0 RESUME                   0

1           2 PUSH_NULL
            4 LOAD_NAME                0 (print)
            6 LOAD_CONST               0 ('Hello')
            8 CALL                     1
           16 POP_TOP
           18 RETURN_CONST             1 (None)
```

Get Code Objects

- Built-in `compile()` function - docs
- Turns an AST, string or file into a code object
- Code objects are Python objects that represent bytecode
  - Used internally but exposed to the user
  - Can be evaluated using the `eval()` or `exec()` built-in functions

example_4_compile.py

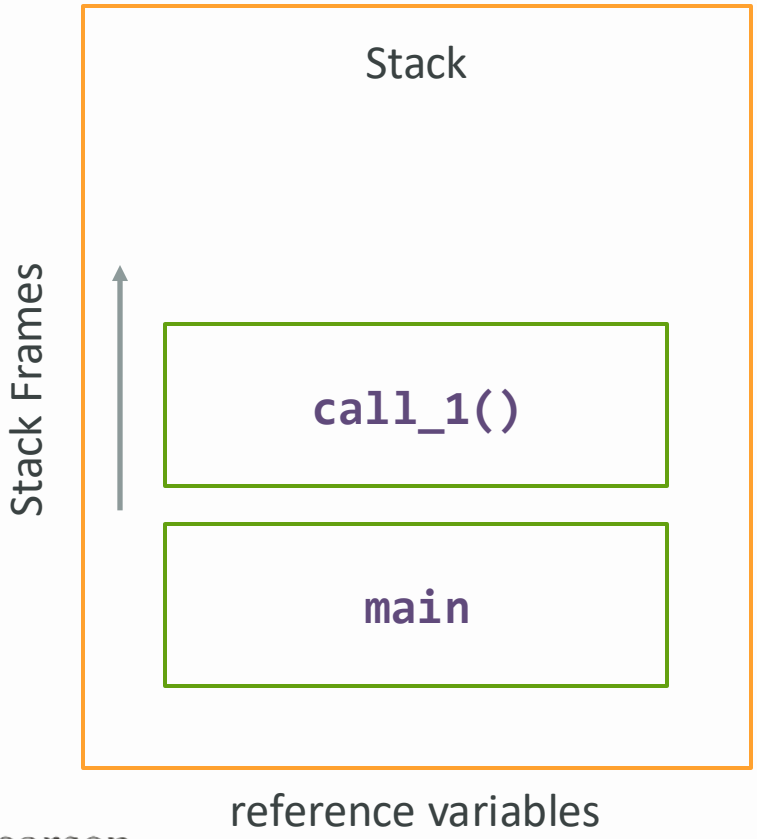
Interpretation

- Bytecode loaded into Python runtime
- A **virtual machine** (written in C) interprets the bytecode, line-by-line, and transforms it into machine code
- Machine code is executed, line-by-line

CPython Interpreter

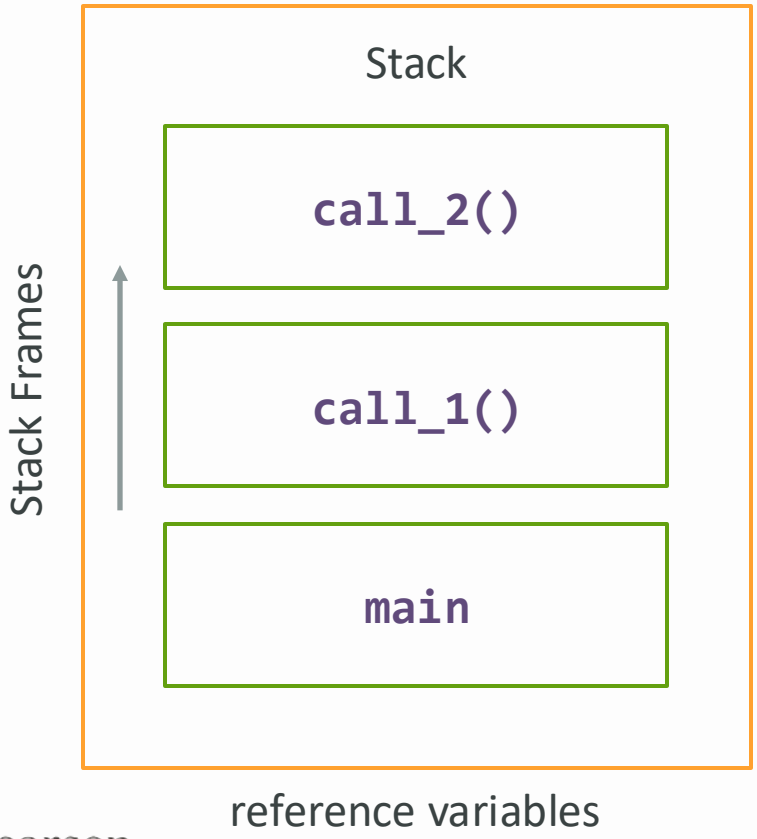
- A.k.a. virtual machine or bytecode interpreter
- Converts bytecode to machine code and executes it
- Implemented in C as a long SWITCH statement on OPCODEs
  - `cpython/Python/ceval.c` (< 3.11)
  - Now `cpython/Python/generated_cases.c.h` is generated from `cpython/Python/bytecodes.c` (as of 3.12)
- Uses an execution stack to keep track of variables in scope

Execution Stack



- Uses frame objects to execute code objects (retrieved from bytecode)
- **Frame objects** contain:
  - namespaces needed (local, global, builtin)
  - An evaluation/value stack
  - A block stack (used to handle loops and exceptions)

Execution Stack



- Frame objects are added to the execution stack when a new function is called

CPython Interpreter vs JIT

- **CPython**'s interpreter:
  - Looks at each line of bytecode
  - Checks the big switch statement for the OPCODE
  - Executes the instructions

- A **JIT compiler** will:
  - Analyzes the code
  - Sometimes uses the switch statement
  - Sometimes it translates code blocks into a sequence of machine code
  - Can create much more optimized machine code
  - Longer up-front cost, faster execution for longer programs

Resources – CPython Overview

- [Medium] **A Tiny Guide to the CPython Interpreter**
  - https://medium.com/@sudarshbuxyyes/a-tiny-guide-to-the-cpython-interpreter-2556d977b6f8

- [PyCascades 2023] **Why 'Hello World' is a Massive Operation**
  - https://www.youtube.com/watch?v=RcGshw0tzoc

- [PyCon 2018] **The AST and Me**
  - https://www.youtube.com/watch?v=XhWvz4dK4ng

Resources – Compiler Steps

- [Brown Water Python] **Better Docs for the Python tokenize Module**
  - https://www.asmeurer.com/brown-water-python/

- [Green Tree Snakes] **the missing Python AST docs**
  - https://greentreesnakes.readthedocs.io/

- [Python] **Disassembler for Python bytecode**
  - https://docs.python.org/3/library/dis.html

Resources - JIT

- [RealPython] **PyPy: Faster Python With Minimal Effort**
  - https://realpython.com/pypy-faster-python/

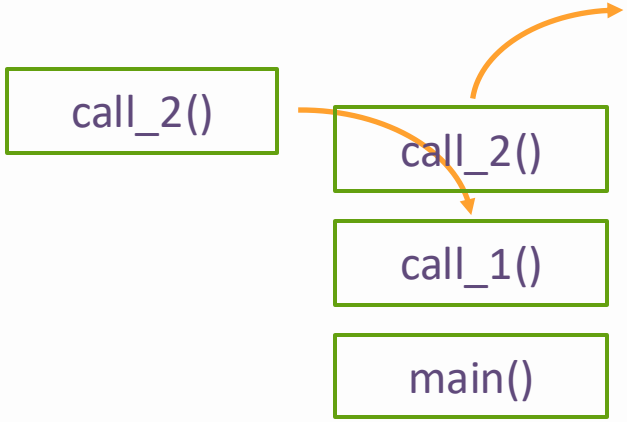
- [Numba] **JIT compiler for Python and NumPy**
  - https://numba.pydata.org/

- [Anthony Shaw] **Python 3.13 gets a JIT**
  - https://tonybaloney.github.io/posts/python-gets-a-jit.html

Memory management

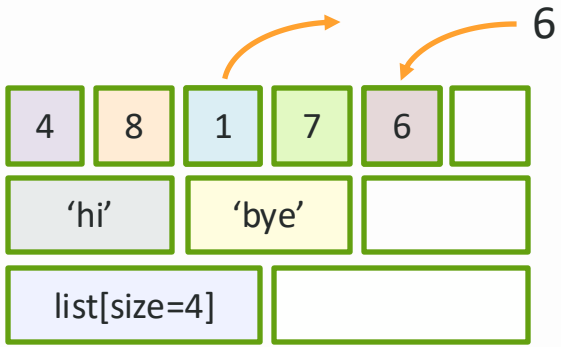
Types of Memory Assignment

- **Stack memory**
  - Represents function calls with local variables
  - When a function is called, a new frame is added to the stack
  - When a function returns, the memory in that scope may be freed
  - Fast, limited, short-term storage

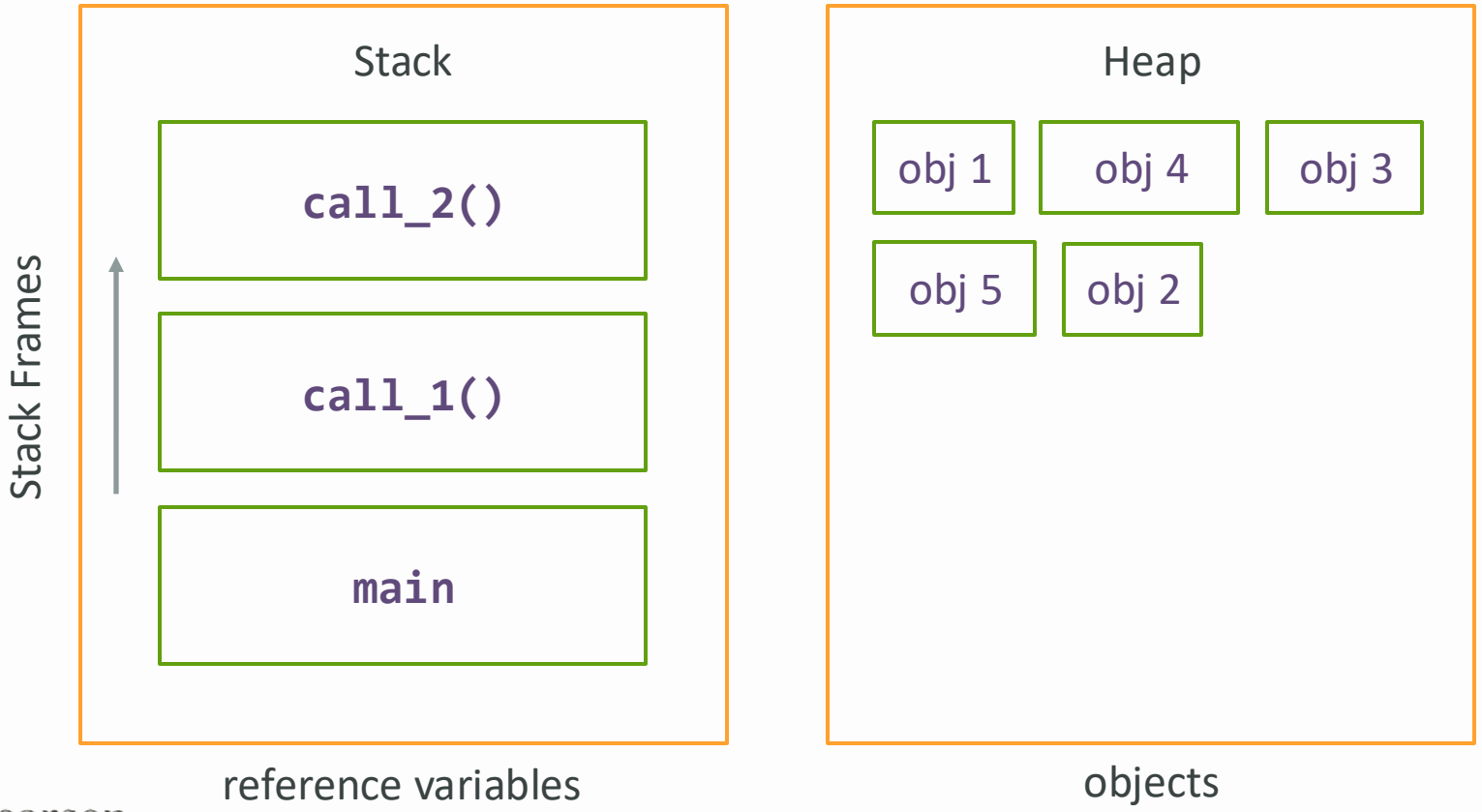


Types of Memory Assignment

- **Heap memory**
  - Like lists of blocks of different sizes
  - Objects get blocks allocated to them when created
  - Memory can get freed via a trigger or when program ends
  - Slower, larger, long-term storage



Memory


Stack
call_2()
call_1()
main
Stack Frames
reference variables
Heap
obj 1
obj 4
obj 3
obj 5
obj 2
objects

Variable assignment

```
x = "Hello, world"
```

- Creates a new object in the heap, with:
  - type = `str`
  - value = “`Hello, world`”
- In the topmost stack frame:
  - **x** is saved as a local variable name
  - **x** set to point to the object in the heap

Variable assignment

```
x = "Hello, world"
```

- Creates a new object in the heap, with:
  - type = `str`
  - value = “`Hello, world`”
- In the topmost stack frame:
  - **x** is saved as a local variable name
  - **x** set to point to the object in the heap

```
y = x
```

- Variable **y** stored on the stack, pointing to the same object

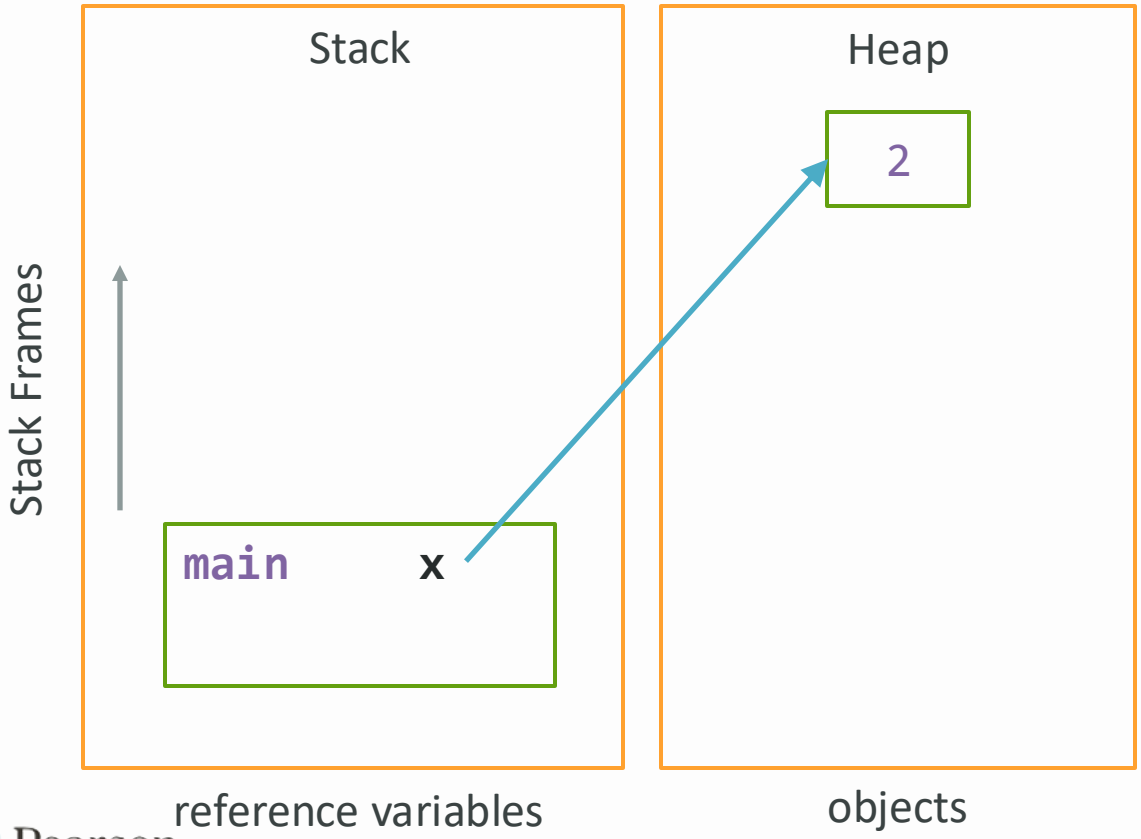
Memory example

- Given a program

```
def func_2(x):
    x = x - 1
    return x

def func_1(x):
    x = x * 2
    y = func_2(x)
    return y

x = 2
y = func_1(x)
```

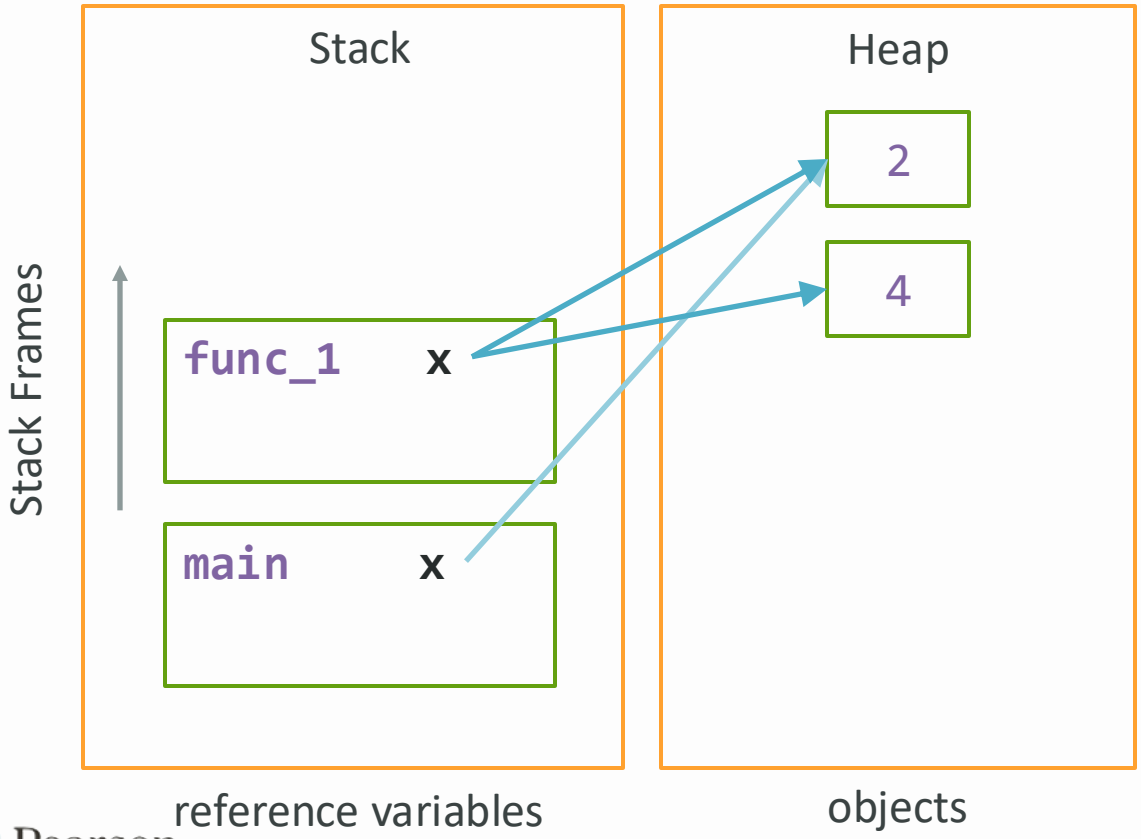
example_5_memory.py

Memory example



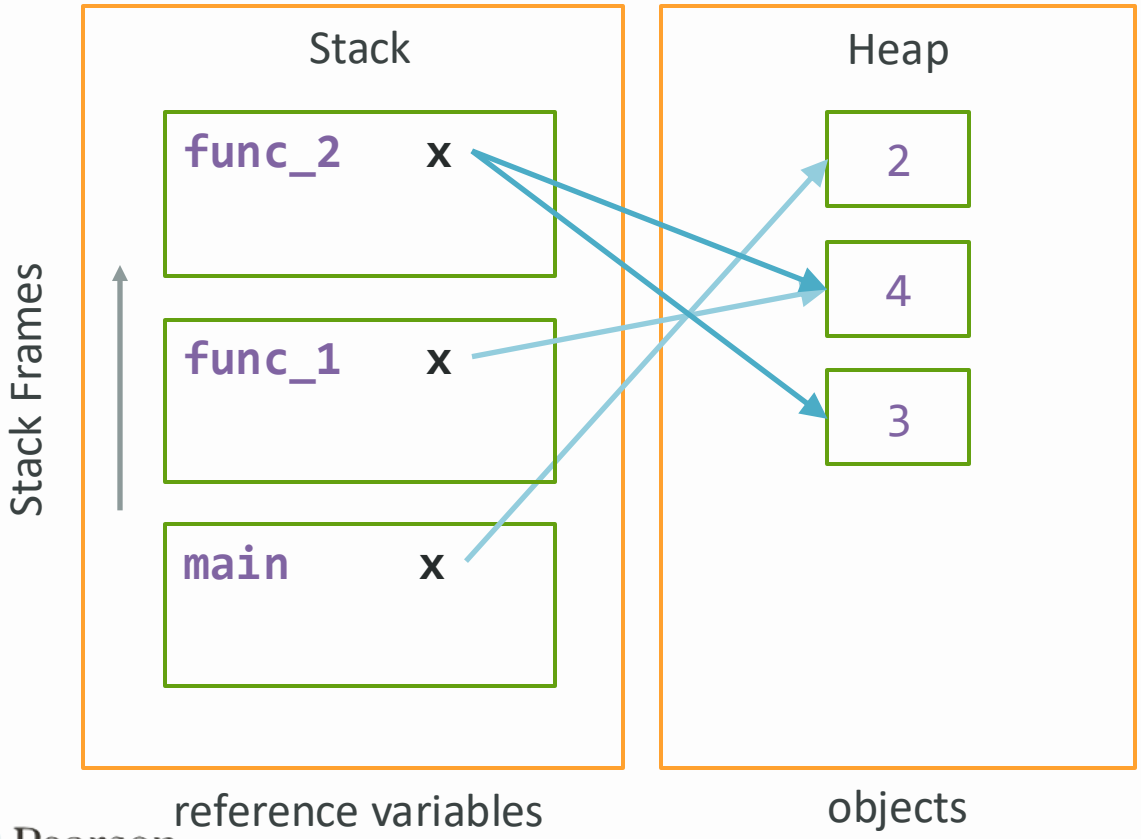

```
def func_2(x):
    x = x - 1
    return x

def func_1(x):
    x = x * 2
    y = func_2(x)
    return y

x = 2
y = func_1(x)
```

Memory example



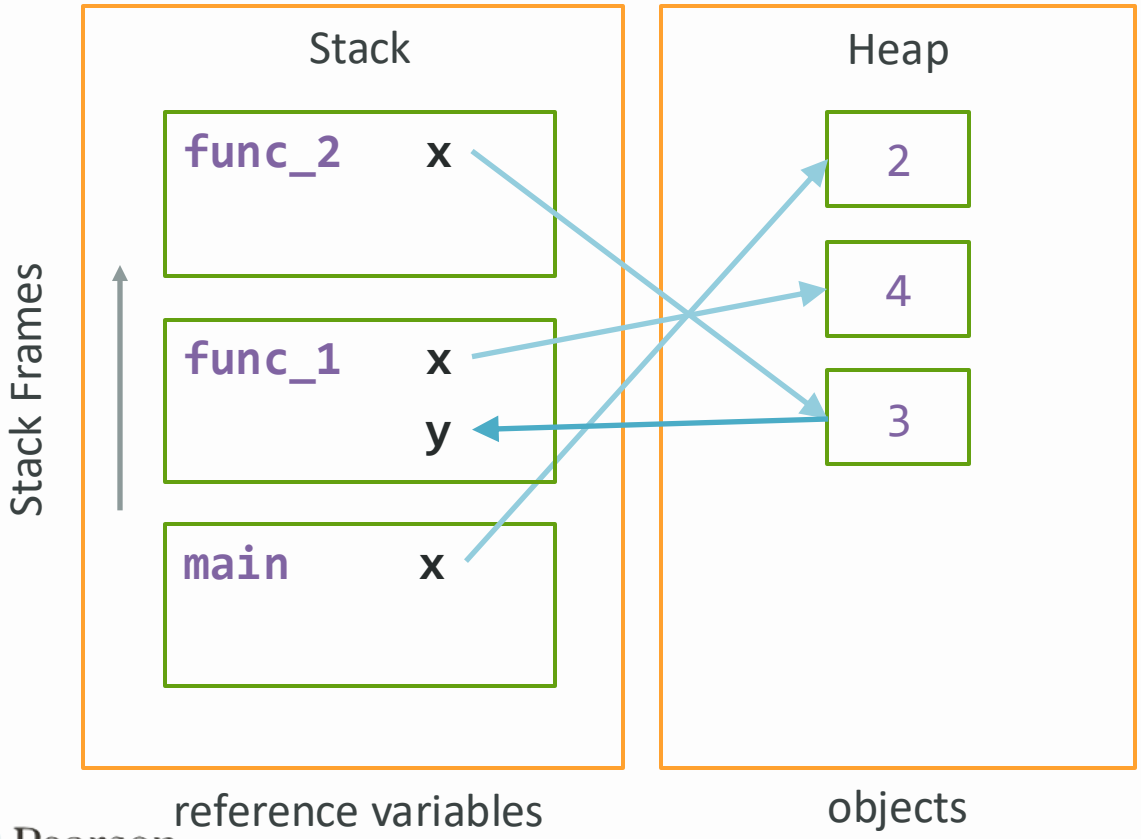

```
def func_2(x):
    x = x - 1
    return x

def func_1(x):
    x = x * 2
    y = func_2(x)
    return y

x = 2
y = func_1(x)
```

Memory example



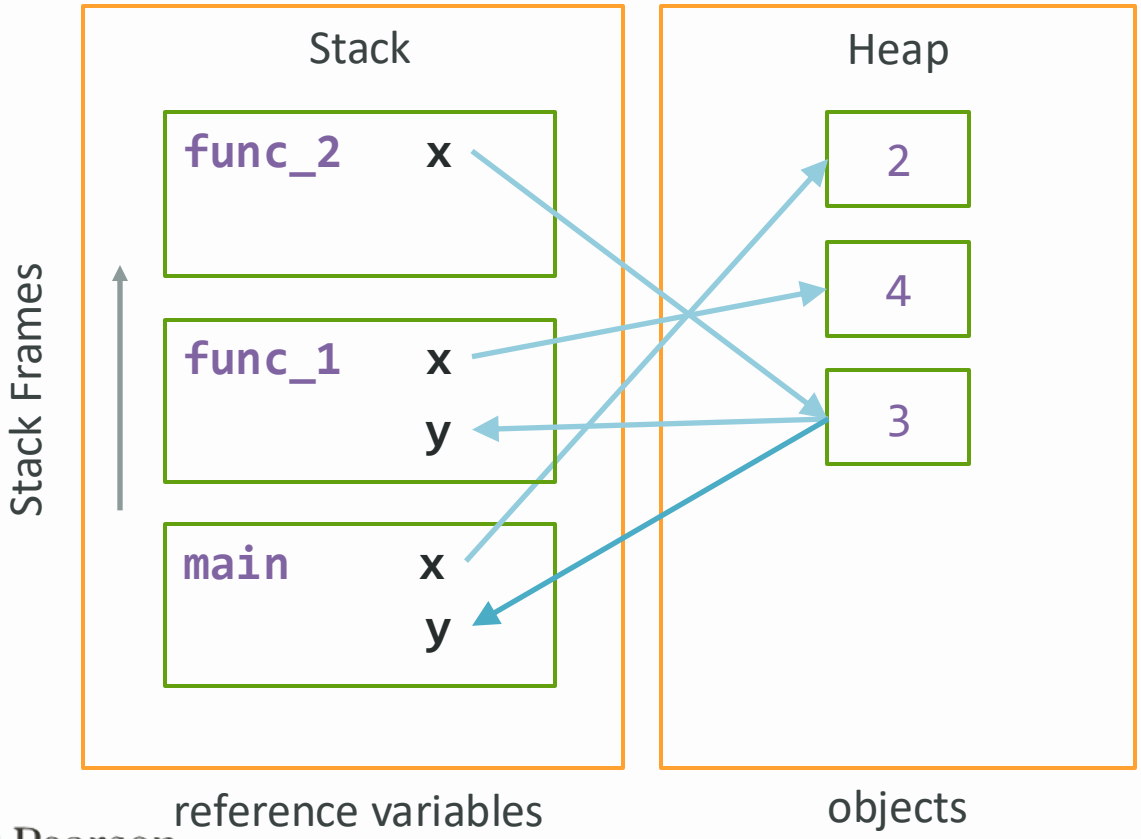

```
def func_2(x):
    x = x - 1
    return x

def func_1(x):
    x = x * 2
    y = func_2(x)
    return y

x = 2
y = func_1(x)
```

Memory example



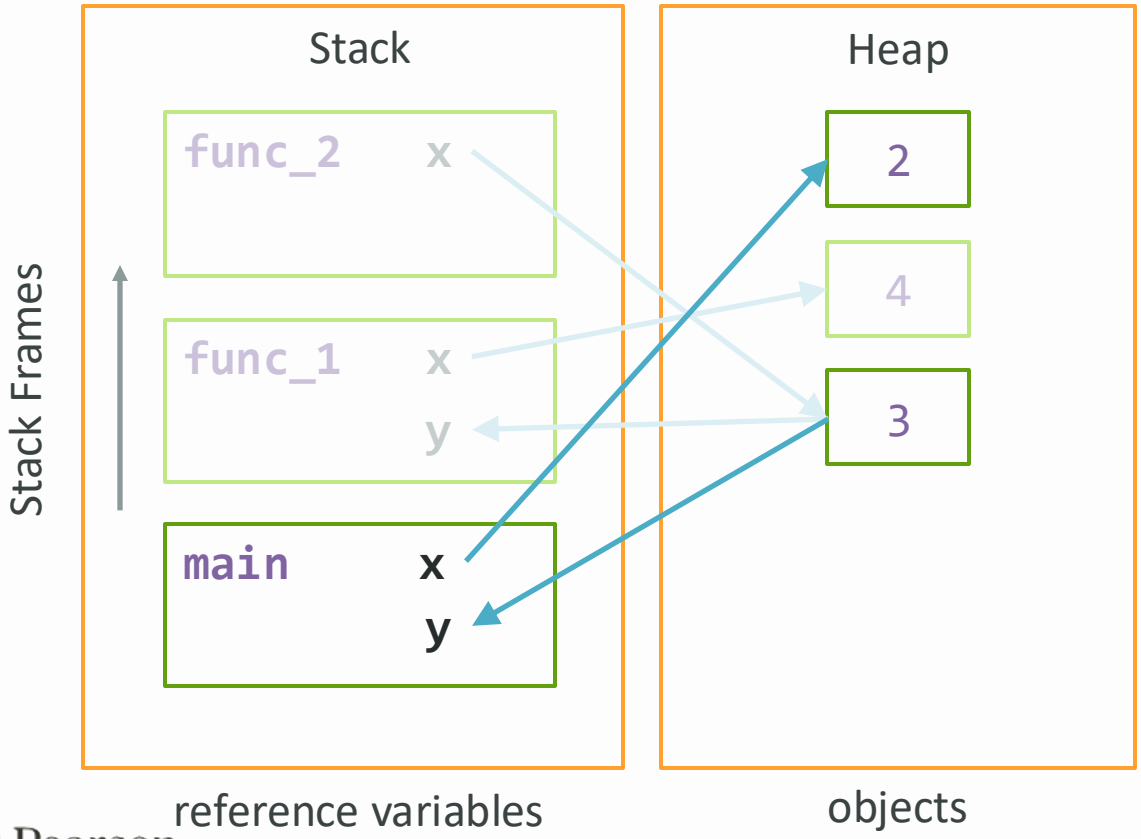

```
def func_2(x):
    x = x - 1
    return x

def func_1(x):
    x = x * 2
    y = func_2(x)
    return y

x = 2
y = func_1(x)
```

Memory example




```
def func_2(x):
    x = x - 1
    return x

def func_1(x):
    x = x * 2
    y = func_2(x)
    return y

x = 2
y = func_1(x)
```

Memory example



- Frames for `func_2` and `func_1` are removed from the stack.
- Local variable pointers are freed.
- **4** in heap memory can be cleared.

Memory Management in Python

- **Reference counting**
  - Frees memory when an object has no more references
- **Garbage collection**
  - Frees objects, with reference count > 0, that are unreachable

Reference Counting

- CPython keeps track of the # of references to an object
- If references drop to 0, the memory is released
- Check the number of references using:

```
import sys
print(sys.getrefcount(obj))
```

  - Note: this will give you +1 because of passing it into the function
  - If larger than expected, it's being referenced by Python internally or is marked as immortal (e.g. 4294967295, new in 3.12)

PyObject

- In CPython, every object is a PyObject.
- These are the objects stored in the heap.

- **PyObject** contains:
  - A value
  - A reference count
  - A pointer to its type object

example_6_refcount.py

Garbage Collection

- Reference counting can’t free objects that have references to themselves (cyclical references).
- **Garbage collector** occasionally runs and checks if objects are reachable. If not, it releases them.
- Can be modified, manually run, and disabled.
  - How Instagram improved efficiency by disabling GC

example_7_cyclical_refs.py

Global Interpreter Lock (GIL)

- Only **one thread** can control the Python Interpreter at a time
  - Single-threaded programs not affected
  - Bottleneck on CPU-bound, multi-threaded code

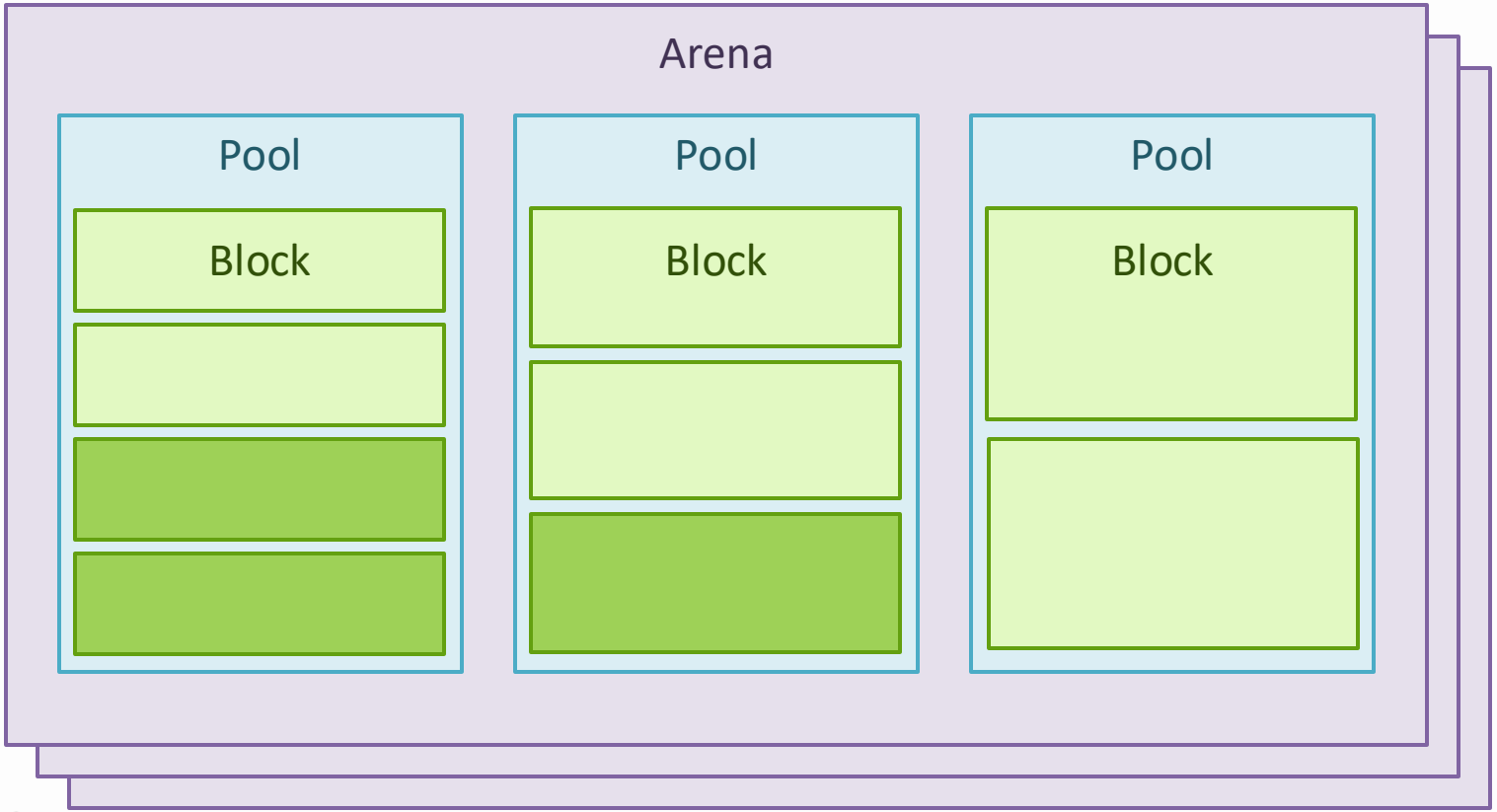
- Needed to ensure reference counting works
  - Race conditions won't accidentally free memory
  - Prevents writing to the same memory location

- GIL will hopefully be optional in the future (PEP 703)

Free-threaded mode

- Python 3.13 added experimental support for disabling the GIL
- Expect bugs and a single-threaded performance hit
- Running in free-threaded mode
  - Requires a different executable (usually called python3.13t)
  - Or build CPython from source with --disable-gil option

Python's heap memory



Resources - Memory

- [Computer Science Wiki] **Heap memory**
  - https://computersciencewiki.org/index.php/Heap_memory

- [RealPython] **Memory Management in Python**
  - https://realpython.com/python-memory-management/

- [PyCon 2016] **Memory Management in Python - The Basics**
  - https://www.youtube.com/watch?v=F6u5rhUQ6dU

Resources – Garbage collection

- [Python Developer's Guide] **Garbage collector design**
  - https://devguide.python.org/internals/garbage-collector/

- [Stackify] **Python Garbage Collection: What It Is and How It Works**
  - https://stackify.com/python-garbage-collection/

Resources - GIL

- [RealPython] **What Is the Python Global Interpreter Lock (GIL)?**
  - https://realpython.com/python-gil/

- [testdriven.io] **Parallelism, Concurrency, and AsyncIO in Python - by example**
  - https://testdriven.io/blog/python-concurrency-parallelism/

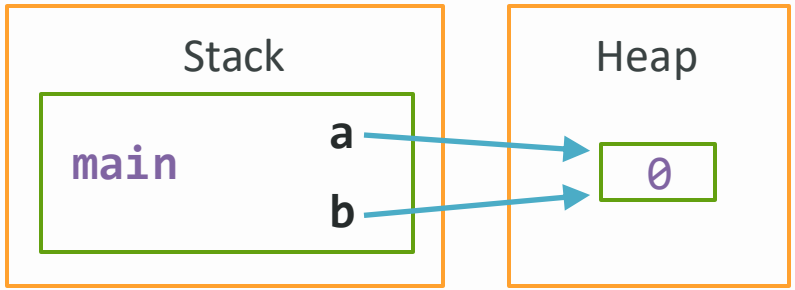
Namespace and scope

- **Supplemental video:**
  - Next Level Python
    - 5.3 Learn about namespaces and scope in Python

- **Interactive lab:**
  - Modules and Packages

Variable assignment

```
a = 0
b = a
```



- Equivalent to

```
a = b = 0
```

- Object stored in the heap
- Variable pointer stored in the stack
- Multiple variables can point to the same object

Scope

- Built-in, global, local
- Each has own mapping of name → object, stored as `dict`
- To find a name:
  - Checks locals
  - Checks locals of any outer functions (enclosed)
  - Checks globals
  - Checks builtins
- Called the LEGB rule

example_8_scope.py

Scope

- Assigning to variables adds them to the current local scope.
- If there are name conflicts, it will follow the LEGB rule.
- Mutable global objects can be mutated within a function.

global

- If you want to reassign a global variable in local scope, use the `global` keyword.
- Marks a name as coming from the global scope.
- Doesn’t add it to the local scope.
- Not needed if mutating a global mutable collection.
- Using this is a bad practice. Don’t do it!

example_9_global.py

```
num_calls = 0

def a_function():
    global num_calls
    num_calls += 1

a_function()
a_function()
a_function()
print(num_calls)  # 3
```

nonlocal

- Similar to global, but used to reference a variable in an enclosed scope (outer function).

- Useful for **closures**:
  - Outer function returns an inner function.
  - Inner function can retain variables/state of outer function, even after the outer function goes out of scope.

example_10_nonlocal.py

Scope Best Practices

- Use unique names, no matter what scope you're in.
  - Includes packages, modules and built-in functions
- Use local names rather than global names.
- Write self-contained functions.
- Avoid global name modifications.
  - Don't modify mutable global variables.
- https://realpython.com/lessons/preventing-scope-pitfalls/

Refactored

```
num_calls = 0

def a_function():
    global num_calls
    num_calls += 1
```

becomes

```
num_calls = 0

def a_function(call_count):
    return call_count + 1

num_calls = a_function(num_calls)
```

Resources – Namespace and scope

- [RealPython] **Python Scope & the LEGB Rule**
  - https://realpython.com/python-scope-legb-rule/

- [Python in Plain English] **Mastering Variable Scope in Python**
  - https://python.plainenglish.io/understanding-variable-scope-in-python-a-comprehensive-guide-to-understanding-managing-and-a7b72cd18904

Everything is an object

Variables

Variables in C

```
int x = 123;
```

x	
Location	4399277360
Value	123

```
x += 1;
```

x	
Location	4399277360
Value	124

← No change

Variables in C

x	
Location	4399277360
Value	124

```
int y = x;
```

y	
Location	4399277392
Value	124

← Change

Variables in Python

```
x = 123
```

x	
Location	4399277360
Value	123

```
x += 1
```

x	
Location	4399277392
Value	124

← Change

Variables in Python

x	
Location	4399277392
Value	124

```
y = x
```

y	
Location	4399277392
Value	124

← No change

Variables in Python

- Variables in Python are also called names.
- Names live on the stack, while the objects they point to live in the heap.
- A **namespace** is a mapping of names to objects.
- Python has namespaces for: built-ins, globals and locals.

What is an object

Objects in Python

- Everything is an object
  - Instances, classes, functions, types, methods, `True`, `None`
  - `isinstance(x, object)` always returns `True`
- Every **`object`** has:
  - An id
  - A type
  - A value
- Objects are stored in heap memory
- References to objects are stored in stack memory

ID

- **ID**
  - Use `id()` to see an `int` representing its identity
  - In CPython, this is its memory address
  - Cannot change

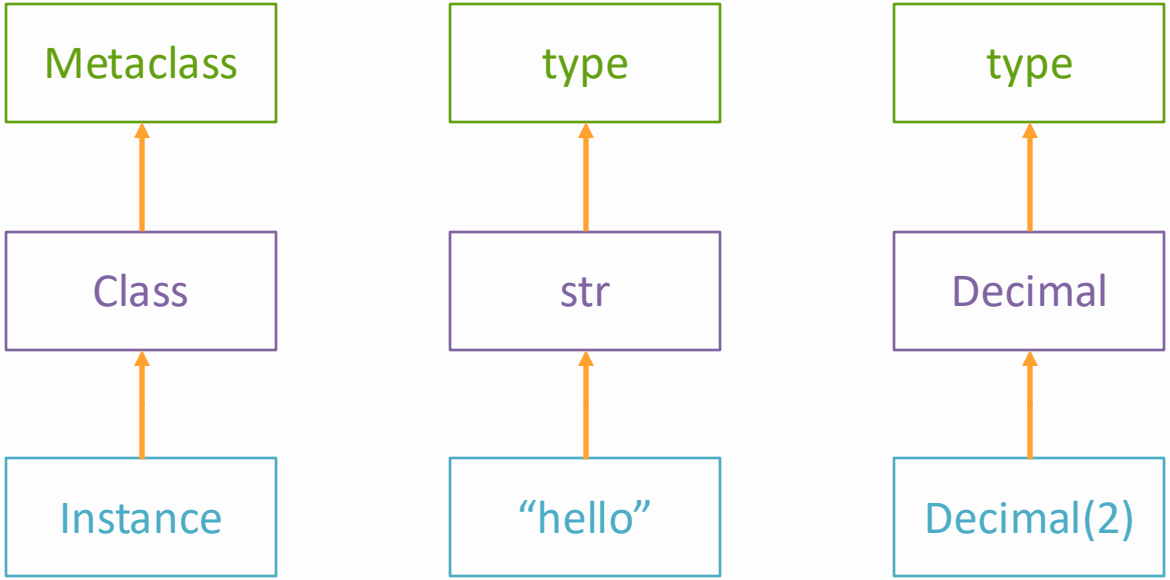
Type

- **Type**
  - Use `type()` to see its type
  - Determines the methods and possible values for an object
  - Cannot change

Value

- **Value**
  - Mutable objects
    - Contents can change
    - E.g. `list`, `dict`, `set`
  - Immutable objects
    - Value cannot change
    - E.g. `int`, `str`, `tuple`

Type Hierarchy of Objects



type()

- Calling `type()` on an object will return a class one level up in the hierarchy

- `type("hello")` returns `<class 'str'>`
- `type(str)` returns `<class 'type'>`

- type itself is actually a class, not a function
- `type(type)` returns `<class 'type'>`

- `isinstance(obj, cls)`, `issubclass(cls, cls)`

Class vs Type?

- Type and class are fairly interchangeable.
- *Type* is an abstract concept, while *class* is an instance of a type.
- A *class* is a Python object you can interact with.

Old-style Classes

- Prior to Python 2.2:
  - *Type* referred only to built-in types (e.g. int, bool, str, list)
  - *Class* referred to user created classes (e.g. Decimal, User)

- Old-style classes didn't inherit from `object`

```
class User:
    pass
user = User()
```

  - `type(user)` returned `<type 'instance'>`
  - `type(User)` returned `<type 'classobj'>`

New-style Classes

- In Python 2.2+, new-style classes had to inherit from `object`:

```
class User(object):
    pass
user = User()
```

  - `type(user)` returned `<class 'User'>`
  - `type(User)` returned `<class 'type'>`

- New style classes are what we use now.
- In Python 3, you don't need to inherit from `object` explicitly.

Example

Online Python 2 interpreter

- https://onecompiler.com/python2/422badyzx

Resources – CPython

- [Python] **Data Model**
  - https://docs.python.org/3/reference/datamodel.html
- [Python C-API] **Common Object Structures**
  - https://docs.python.org/3/c-api/structures.html
- [Python] **Unifying types and classes in Python 2.2**
  - https://www.python.org/download/releases/2.2.3/descrintro/

ID, equality, containers

ID

- All objects have a unique id
- Variables that point to the same object will have the same id

- `is` checks if id's are equal
- `==` checks if values are equal

```
x = True
y = True
x is y           # True
id(x) == id(y)   # True
```

```
x = [True]
y = [True]
x is y           # False
id(x) == id(y)   # False
```

ID – Immutable objects

```
x = 'hello'
y = 'hello'
x is y          # True
```

- x and y may point to the same object after

- An optimization in CPython (immortalization)
  - Commonly used objects may get re-used
  - Reference count never changes, never deleted from memory
  - None, `True`, `False`, integers [-5, 256], and single-word strings

```
sys.getrefcount(-5)    # 4294967295
```

ID – Mutable objects

```
x = []
y = []
x is y          # False
```

- x and y are mutable, so won't point to the same object unless explicitly stated

```
x = []
y = x
x is y          # True
```

- or

```
x = y = []
x is y          # True
```

Mutable vs Immutable Objects

- Modifying an immutable objects returns a new object

```
x = 5
id(x)  # 4354299504

x += 1
id(x)  # 4354299536
```

- Modifying a mutable object doesn't

```
x = [5]
id(x)  # 4362298432

x += [1]
id(x)  # 4362298432
```

Variable reassignment

```
x = 'hello'
y = x
x is y          # True
```

- x and y are references (in the stack) to objects (in the heap)
- **`y = x`**, it creates a new name y that reference the same object as x

```
x = 'goodbye’
x is y          # False
```

- x references a new object (‘goodbye’)
- y still references the old object (‘hello’)

- No way to link **x** and **y** so that when one changes, the other changes

Containers

- Contain references to other objects

- `list`, `tuple`, `dict`, `set`, etc…

```
x = []
y = x

y.append(1)
print(x)    # [1]
```

- x and y refer to the same mutable object
- Changes to x reflect on y

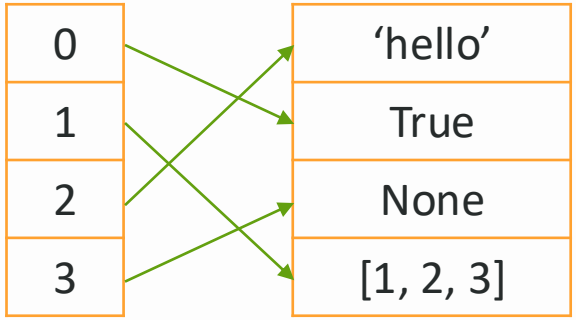
Containers

```
x = []

y1 = (x, )
y2 = (x, x)

x.append(1)
print(y1)  # ([1], )
print(y2)  # ([1], [1])
```

- Different containers can contain references to the same mutable object

Containers

- Contents of containers are references to other objects
- Any container can hold any kind of object



- Containers can even reference themselves
- Use `array.array` or `numpy.ndarray` (etc) for more memory efficient arrays that can only hold one type of data

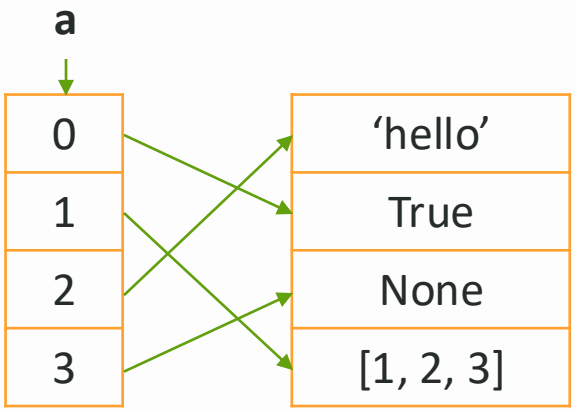
example_11_variables.py

Copying containers and objects

```
import copy

a = [[]]
b = a
c = copy.copy(a)
d = copy.deepcopy(a)

a.append(1)
a[0].append(2)
print(a)  # [[2], 1]
print(b)  # [[2], 1]
print(c)  # [[2]]
print(d)  # [[]]
```

- `copy.copy` makes a new object and shallow copies contents
- `copy.deepcopy` makes copies of all contents, recursively

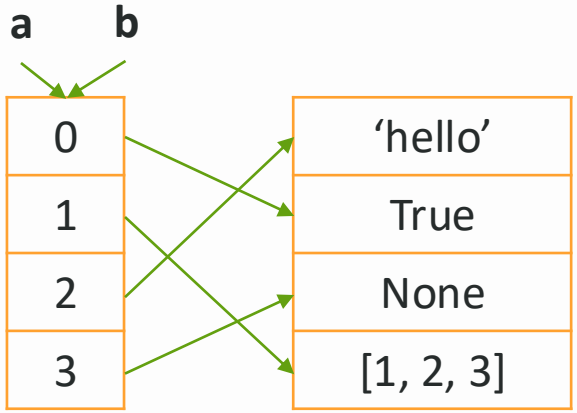
Copy



- Variable reassignment
- `copy.copy`
- `copy.deepcopy`

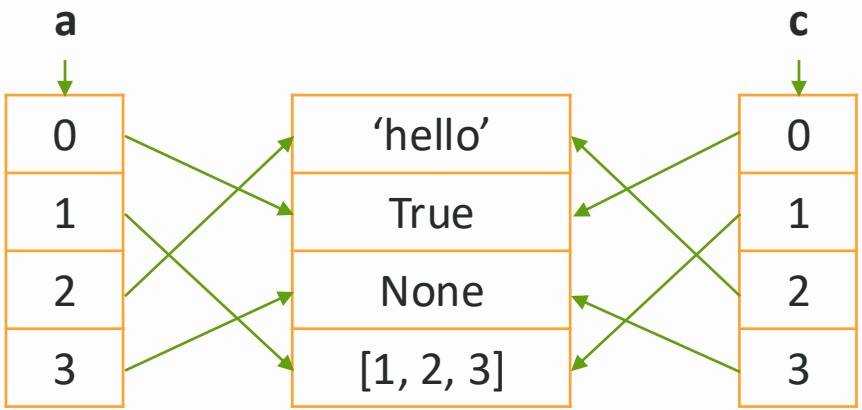
Variable assignment

- Variable reassignment assigns new name to the existing object



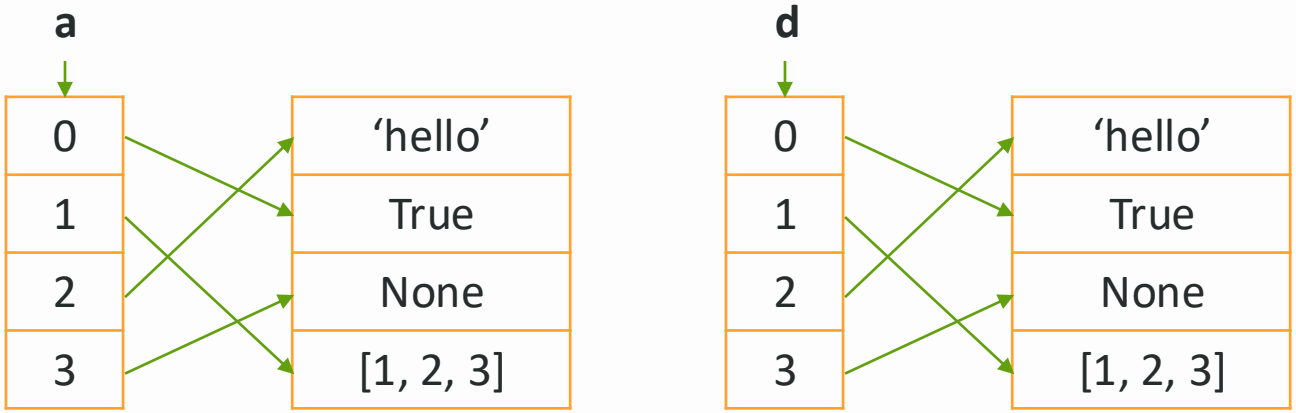
copy.copy

- copy.copy creates a new object, pointing to the existing contents



copy.deepcopy

- copy.deepcopy creates a new object, pointing to new objects



- **Note**:
  - References to immutable objects may still point to the same object (increasing their reference count)
  - References to mutable objects will always point to new objects

example_12_copy.py

Question

```
def append_7(a_list=[]):
    a_list.append(7)
    return a_list

append_7()
print(append_7())
```

- What would print to the console?

A. []
B. [7]
C. [7, 7]

It's a value, not an expression!

Resources – Python variables

- [Python] **Data Model**
  - https://docs.python.org/3/reference/datamodel.html

- [RealPython] **Pointers in Python**
  - https://realpython.com/pointers-in-python/

Inspecting objects

help()

- `help()`
  - Calling `help()` in the Python console starts an interactive help session
  - Calling `help(thing)` prints help for the Python object 'thing'

```
>>> help(list)
Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return bool(key in self)
```

vars()

- `vars()`
  - Without arguments, equivalent to `locals()` (or `globals()`)
    - Return a dictionary containing the current scope's local variables
  - With an argument, equivalent to `object.__dict__`

```
>>> pprint(vars(list))
mappingproxy({'__add__': <slot wrapper '__add__' of 'list' objects>,
              '__class_getitem__': <method '__class_getitem__' of 'list' objects>,
              '__contains__': <slot wrapper '__contains__' of 'list' objects>,
              '__delitem__': <slot wrapper '__delitem__' of 'list' objects>,
              '__doc__': 'Built-in mutable sequence.\n'
                         '\n'
                         'If no argument is given, the constructor creates a '
                         'new empty list.\n'
                         'The argument must be an iterable if specified.',
              '__eq__': <slot wrapper '__eq__' of 'list' objects>,
```

`__dict__`

- A dictionary or other mapping object used to store an object's attributes.

- Objects that can have **`__dict__`**:
  - **Modules** – its global namespace
  - **Class** – class methods and attributes
  - **Instance** – instance attributes
  - **Function** – function attributes (see PEP 232 for rationale)

dir()

- `dir()`
  - List of attributes of an object.
  - Without an argument, returns names in the current scope.
  - Else, returns a sorted list of names of the attributes of the object, including attributes on its class and base classes.

```
>>> pprint(dir(list))
['__add__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
```

repr()

- `repr()`
  - Return the “official” string representation of an object.
  - Ideally, this should look like a valid Python expression.
  - It could be used to recreate an object with the same value.

- `repr('hi')` returns `"'hi'"`
- `repr(['a', 'b’])` returns `"['a', 'b’]"`
- `repr(exc)` returns `'NameError("name \'x\' is not defined")’`
- `repr(print)` returns `'<built-in function print>'`

`inspect` module

- The built-in **`inspect`** module has functions to get information about objects and code.

- **`getmembers(object[, predicate])`**
  - Get a list of (attribute, value) pairs for an object.
  - **predicate** is an optional function to filter the values by.
    - E.g. `inspect.getmembers(1, inspect.isroutine)`
    - `inspect.getmembers(1, lambda x: not inspect.isroutine(x))`

- **`getsource(object)`**: Get source code as string
- **`getsourcelines(object)`**: Get source code as a list of strings

example_13_vars_dirs.py

Attributes

- Any reference you can access via dot notation
  - instance.attr
  - instance.method
  - Class.attr
  - Class.method
  - module.function
  - module.variable
  - package.module

Using Attributes

- Get
  - `obj.attr_name`
- Set
  - `obj.attr_name = value`
- Delete
  - `del obj.attr_name`
- Custom classes:
  - Getting or deleting attributes that don't exist raises an `AttributeError`.
- Built-in types:
  - Setting or deleting attributes raises an `AttributeError`.

Attr Functions

- Built-in functions allow you to work with names as strings:
  - Get
    - `getattr(obj, 'attr_name')`
  - Set
    - `setattr(obj, 'attr_name', value)`
  - Containment check
    - `hasattr(obj, 'attr_name')`
  - Delete
    - `delattr(obj, 'attr_name')`

Duck typing

- “If it walks like a duck and it quacks like a duck, then it must be a duck.”

- We often care more about an object’s behaviour, than its actual type.

- Can the object be called -> callable
- Does it support iteration -> iterable

Duck typing

- Instead of type checks with `isinstance(obj, type)`,
- we can use attribute checks with `hasattr(obj, 'attr_name')`

- Or, attempt to access an attribute in a `try/except` block, handling `AttributeErrors`

- `callable(obj)` checks if an object can be called

- https://realpython.com/duck-typing-python/

example_14_attrs.py

Resources – Object Introspection

- [Python] **Built-in Functions**
  - https://docs.python.org/3/library/functions.html
- [Python] **`inspect` — Inspect live objects**
  - https://docs.python.org/3/library/inspect.html
- [Devopedia] **Introspection in Python**
  - https://devopedia.org/introspection-in-python
- [Anvil] **Introspection in Python**
  - https://anvil.works/blog/introspection-in-python

Debuggers and IDE's

Poll (multi-choice)

- Which IDEs do you use for Python
  - VS Code
  - PyCharm
  - Jupyter notebooks
  - Spyder
  - IDLE
  - SublimeText
  - Terminal-based editors (vim, emacs, nano, …)
  - Other (say in chat)

Tips for working with objects

- Functions to use:
  - `help()`
  - `vars()`, `locals()`, `globals()`
  - `dir()`
  - `repr()`, `print()` and `pprint()`
  - `inspect` module

- Other tools:
  - Debuggers
  - Python Console
  - Type hinting
  - Documentation

Debuggers

- Run a program, but pause when a breakpoint is reached.

- Once paused, you can:
  - Navigate the **execution stack**
  - See **variables** in various scopes
  - **Evaluate** code with the current context
  - **Execute** lines of code (step into, step over, step out)
  - **Resume** until next breakpoint

Debuggers

- **Breakpoints**
  - When the program reaches this line, stop
- **Resume**
  - Run program until another breakpoint is reached
- **Step over**
  - Evaluate the current line and then stop
- **Step into**
  - Go into the evaluation of the current line
- **Step out**
  - Evaluate until the current function and then stop

pdb

- Built-in module for Python debugging

- To use, include:
  - `import pdb; pdb.set_trace()` or
  - `breakpoint()` new in 3.7

- Or run:
  - `python –m pdb <filename.py> [args…]`

pdb commands

Command	Description
p or **pp**	Print or pretty-print the value of an expression
n	Execute until the next line is reached (step over)
s	Execute the next statement (step into)
c	Continue until the next breakpoint (resume)
unt <line>	Continue until line number is reached
l or **ll**	List the source code
w	Print the stack frame
h <topic>	See all commands; if topic provided, show help for it
q	Quit and exit

Try out `pdb`

- Add `breakpoint()` to code
- Show code: `l`
- View variables: `vars()`
- Execute next line: `n`
- Step into function: `s`
- Continue to next breakpoint: `c`

IDE debuggers

- PyCharm
- VS Code + Python plugin
- Jupyter-lab
- Spyder
- Eclipse + PyDev

Debuggers - PyCharm

- PyCharm only shows an object's data attributes by default (not methods)
- To see a list of methods, see code suggestions for `object.`
- To see all methods and vars for an object, evaluate:
  - `{name: getattr(obj, name) for name in dir(obj)}`

Python Console

- Default Python console doesn't have many features
- PyCharm and Spyder have built-in Python Consoles
- **rich** – colour output, better errors, + more
- **IPython** – autocomplete, syntax highlighting, block history
- **bpython** – autocomplete, syntax highlighting, help text
- **ptpython** – autocomplete, syntax highlighting, block history
- 4 alternatives to the standard interactive Python shell

Rich demo

```
$ pip install rich
$ python -m rich

>>> from rich import print
>>> print(vars())
```

Type Hinting

- Introduced in PEP 484, added to Python in 3.6

- Allows for static type checking in Python code

```
def greet(name, shout=False):
    ...
```

becomes

```
def greet(name: str, shout: bool = False) -> str:
    ...
```

Type Hinting

- Variable:

```
vancouver: tuple = (45.63, -122.67)
```

- Collection w/ contents:

```
dublin: tuple[float, float] = (40.10, -83.11)
```

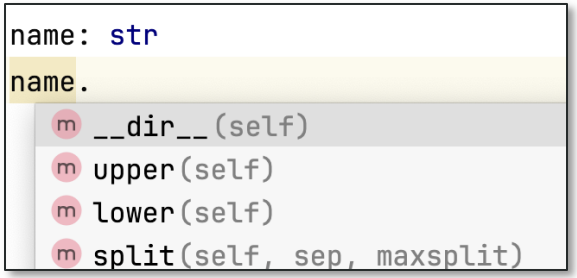
```
from typing import List, Tuple
cities: List[Tuple[float, float]] = [vancouver, dublin]
```

- Type aliases:

```
Coordinates = Tuple[float, float]

cities: List[Coordinates] = [vancouver, dublin]
```

Type Hinting

- Many IDEs can use type hints for better code completion



- PyCharm will warn you when there's a type mismatch

```
round('1.2')
```

- VS Code has a mypy plugin
- Use packages to check your codebase:
  - **MyPy** – original static type checker for Python
  - **Pyre** – Facebook, focuses on performance on large codebases
  - **Pytype** – Google, can infer types and generate annotations

example_15_typehints.py

Documentation

- **Docstrings** – string documentation
  - Used on modules, classes and functions/methods
  - Triple quotes at the beginning of object

- Packages exist for creating API reference sites or automated tests
- See PEP 257 – Docstring Conventions

- **PyCharm**: *F1* shows documentation
- **Jupyter-Lab**: *Shift + Tab* or add *?* to an object and run
- **VS Code**: *Hover* or add *Docs View* extension
- **Console**: `help(obj)`

View Definitions

- Python files – view source code
- C code – view file stubs or documentation

- **PyCharm**: *Ctrl + click* or *Ctrl + B* on an object
- **VS Code**: *Ctrl + click* or *F12* on an object
- **Jupyter**: add *??* to an object and run
- **Console**: `inspect.getsource()`

Resources – Debugging

- [RealPython] **Python Debugging With Pdb**
  - https://realpython.com/python-debugging-pdb/
- [Python] **breakpoint**
  - https://docs.python.org/3.12/library/functions.html#breakpoint
- [PyCharm]
  - **Debugging Python Code**
    - https://www.jetbrains.com/help/pycharm/part-1-debugging-python-code.html
  - **Debug tool window**
    - https://www.jetbrains.com/help/pycharm/debug-tool-window.html

Resources – Console

- [Medium] **Makeover Your Python Terminal with Rich**
  - https://medium.com/analytics-vidhya/makeover-your-python-terminal-14dcc1a1e8a

- [Medium] **4 alternatives to the standard interactive Python shell**
  - https://python.plainenglish.io/4-alternatives-to-the-standard-interactive-python-shell-ff496200b01f

- [IPython] **IPython tutorial**
  - https://ipython.readthedocs.io/en/stable/interactive/tutorial.html

Resources – Type hinting

- [Python] typing - Support for type hints
  - https://docs.python.org/3/library/typing.html
- [RealPython] **Python Type Checking**
  - https://realpython.com/python-type-checking/
- [Bernát Gábor] **The state of type hints in Python**
  - https://bernat.tech/posts/the-state-of-type-hints-in-python/
- [MyPy] **Type hints cheat sheet**
  - https://mypy.readthedocs.io/en/latest/cheat_sheet_py3.html

Useful dunders

The Python Data Model

Data Model

- Python's data design philosophy
  - Documentation

- How Python organizes everything internally
- Basic building blocks of the language itself
- The design/structure of the building blocks
- The basic code-blocks that come into play with them

Objects, Values and Types

- All data in Python is either an object or a relationship between objects

- Every object has an:
  - identity – never changes – use `id(obj)`
  - a type – never changes – use `type(obj)`
  - a value – may change if mutable or stores a mutable object

Objects, Values and Types

- Type determines:
  - the operations that an object supports
  - the possible values for objects of that type
  - whether it is mutable or immutable

- **Containers**: objects that contain references to other objects

- **Mutable** vs. **immutable** containers: can the identities of the immediately contained objects change?

Standard Library Types – 1 / 4

- `None`
- Ellipsis - `...`
- numbers.Number
  - numbers.Integral - `int`
  - numbers.Real - `float`
  - numbers.Complex - `complex`
- Sequences
  - Immutable sequences
    - String - `str`
    - Tuple - `tuple`
    - Bytes - `bytes`

Standard Library Types – 2 / 4

- Sequences cont.
  - Mutable sequences
    - List - `list`
    - Byte Array - `bytearray`
- Set types
  - Set - `set`
  - Frozen set - `frozenset`
- Mappings
  - Dictionary - `dict`

Standard Library Types – 3 / 4

- Callable types
  - User-defined functions
  - Instance methods
  - Generator functions - uses `yield`
  - Coroutine functions - uses `async`
  - Asynchronous generator functions - uses `async` and `yield`
  - Built-in functions
  - Built-in methods
  - Classes
  - Class instances

Standard Library Types – 4 / 4

- Modules
- Custom classes
- Class instances
- I/O objects (files)
- Internal types
  - Code objects
  - Frame objects
  - Traceback objects
  - Slice objects
  - Static method objects
  - Class method objects

Resources – Data Model

- [Python] **Data Model**
  - https://docs.python.org/3/library/typing.html

- [Python in Plain English] **Demystifying Python's Data Model**
  - https://python.plainenglish.io/demystifying-pythons-data-model-59d2edd8fdd5

- [InventWithPython] **The Python Data Model, Explained**
  - https://inventwithpython.com/blog/2018/02/02/the-python-data-model-explained/

Dunders

Dunders

- Reserved names that start and end with ***d***ouble ***under***scores
- Variables

```
if __name__ == "__main__":
```

  - What does this mean?
- Methods

```
class MyClass:
    def __init__(self):
        ...
```

- Python uses this naming convention for its own internal use

Data Model Dunders

- Data model describes:
  - Many dunder variables available on each kind of object
    - E.g. `__name__`, `__dict__`, etc.
  - Which dunder methods are available for classes
    - E.g. `__init__()`, `__eq__()`, etc.
- Not all dunders are described in the data model.
- Anyone can create their own dunders, but in general, you shouldn't.

Dunder files

- `__init__.py`

- **Supplemental video:**
  - Next Level Python
    - 5.2 Understand the use of `__init__.py` files

`__init__.py`

- `__init__.py`
  - If it exists, it explicitly marks the folder it's in as a package that contains Python modules.
  - When a packages is imported, the `__init__.py` file is implicitly run.
  - The objects it defines are added to the package's namespace.

```
# In my_package.__init__.py
x = 5
```

```
# In main.py
from my_package import x
print(x)
```

Dunder functions

- `__import__()`

- **Supplemental video:**
  - Next Level Python
    - 5.1 Look at modules and imports

Dunder functions

- `__import__(name, globals=None, locals=None, fromlist=(), level=0)`
  - Called on `from module import func1, func2`
  - *name* is the module name.
  - *globals* and *locals* are the namespace dicts.
  - *fromlist* contains the names to import.
  - *level* determines if it's an absolute or relative import.

```
_temp = __import__('module', globals(), locals(), ['func1', 'func2'], 0)
func1 = _temp.func1
func2 = _temp.func2
```

Dunder variables

- ~25 dunder variables (Python 3.12)

Dunder variables

- Python conveys information to the programmer (metadata)
  - User can read dunder variables
    - E.g. `__name__`
  - Some dunders are CPython implementation details
    - E.g. `__classcell__`
    - Not covered

- Programmer conveys information to Python
  - Users can write to dunder variables
    - E.g. `__slots__`

Modules 

- `__name__`
  - The module's name. Will be "`__main__`" for the top-level code environment, or the dotted path to the module.
- `__doc__`
  - The documentation string, or None if unavailable
- `__file__`
  - The pathname of the file from which the module was loaded.
- `__annotations__`
  - A `dict` of variable annotations collected during execution. Please see Annotations Best Practices.
- `__dict__`
  - The module's namespace.

Other References

- **Execution model**
  - Documentation
  - How Python structures its programs, binds names, and handles exceptions

- **The import system**
  - Documentation
  - How Python handles importing packages and modules

Modules – not in Data Model

- `__builtins__`
  - A `dict` or the builtins module with all built-in objects
- `__cached__`
  - If `__file__` is set, then location of the cached **.pyc** file
- `__loader__`
  - The loader object that the import machinery used when loading the module. *Deprecated*.
- `__package__`
  - Either the empty string (top-level modules) or the parent package's name (submodules)
- `__spec__`
  - A specification for a module's import-system-related state.

Modules – not in Data Model

- `__all__` 
  - A list of strings of names to be imported from the module on:
    - `from module import *`

example_16_module_dunders.py

Functions 

- `__doc__`, `__name__`, `__annotations__`
- `__globals__`
  - A reference to the `dict` of the function's global variables
- `__closure__`
  - None or a tuple of cells for the function's free variables.
- `__qualname__`
  - Qualified name (including class/outer function name)
- `__module__`
  - The name of the module the function was defined in.
- `__dict__`
  - Holds function attributes. See PEP 232 for rationale.

Functions

- `__defaults__`
  - A tuple containing default parameter values, or None there are no defaults.
- `__kwdefaults__`
  - A dictionary containing defaults for keyword-only parameters.
- `__type_params__` - *New in 3.12*
  - A tuple containing the type parameters of a generic function.
- `__code__`
  - The code object representing the compiled function body.

Instance methods

- `__name__`, `__module__`, `__doc__`
- `__self__`
  - The class instance object the method is bound to.
- `__func__`
  - The original function object.

example_17_function_dunders.py

Code Objects

- Represent byte-compiled Python objects
- Used internally, but exposed to the user
- Definition is specific to interpreter version and can change without notice
- Data model describes definition for each version
- See **example_4_compile.py**

Custom Classes

- `__name__`, `__module__`, `__doc__`, `__annotations__`, `__type_params__`
- `__dict__`
  - The class's namespace
- `__bases__`
  - A tuple of base classes, in order of their occurrence in the base class list.
- `__mro__`
  - A tuple of classes to look at for the method resolution order.

Class Instances 

- `__dict__`
  - The attribute dictionary (on self)

- `__class__`
  - The instance's class

example_18_class_dunders.py

Class Customization Dunders

- `__slots__`
  - Explicitly declare data attributes.
  - `__dict__` and `__weakref__` don't get created.

  Faster than using `__dict__`.

- `__match_args__`
  - Allows for positional arguments in structural pattern matching.

example_19_slots.py

Exceptions

- `__traceback__`
  - The traceback object (stack trace of an exception)

Other dunder variables

- `__weakref__`
  - Set when weak references to an object are made through the weakref module. These are references that are not enough to keep an object alive.
- `__future__`
  - Allow future features to be used in a Python version before it's released. E.g. in Python 2.7
    `from __future__ import unicode_literals`.
- `__objclass__`
  - Exists on a class instance when the class has a descriptor

Resources – Dunder variables

- [Python] **Data Model**
  - https://docs.python.org/3/reference/datamodel.html
- [Python Morsels] **Dunder variables**
  - https://www.pythonmorsels.com/dunder-variables/
- [Python] **Execution Model**
  - https://docs.python.org/3/reference/executionmodel.html
- [Python] **The Import System**
  - https://docs.python.org/3/reference/import.html

Dunder methods

- Special methods (Python docs)
- Magic methods

Question

Given:

```
nums = [1, 2, 3]
nums2 = nums
```

What is the value of nums2 after:

```
nums = nums + [4]
```

Versus:

```
nums += [4]
```

Dunder methods

- Classes can overload the functionality of certain operators.
  - `obj1 + obj2`
  - Will call the `.__add__()` method on obj1's class
  - E.g. `obj1 + obj2` calls `obj1.__add__(obj2)`
- Classes define how operators work on them by implementing the corresponding dunder methods.
- Attempting to execute an operation when the corresponding method isn't defined raises an exception.
  - Usually `AttributeError` or `TypeError`
  - Can set methods to `None` to disable functionality

Calling methods

- Methods can be called in two ways:
  - **`instance.method(arg)`**
    - E.g. `"hello".upper()`
  - **`class.method(instance, arg)`**
    - E.g. `str.upper("hello")`
- Dunder methods written like `object.__str__(self)`
  - May also be written as `self.__str__()`
- If the first argument is `self`, it's an instance method
- If it's `cls`, it's a class method

Basic customization

- `.__new__(cls[, ...])`
  - Constructor; creates a new instance of *cls*.
  - Can be used on regular classes or custom metaclasses.

- `.__init__(self[, ...])`
  - Initializer; set instance attributes here.
  - Called after the instance has been created (by `__new__`), but before it is returned to the caller.

- `an_instance = AClass(arg1, arg2)`
  1. `an_instance = object.__new__('Aclass', arg1, arg2)`
  2. AClass.__init__(an_instance, arg1, arg2)

Basic customization

- `.__str__(self)`
  - Returns the “informal” or nicely printable string representation of an object. Called by `str()`, `print()` and `format()`.
- `.__repr__(self)`
  - Called by `repr()` to compute the “official” string representation of an object. Used for debugging.
- `.__del__(self)`
  - Called when the instance is about to be destroyed. Also called a finalizer or (improperly) a destructor. Why not to use `__del__`

Basic customization

- `x.__lt__(self, y)`: `x < y`
- `x.__le__(self, y)`: `x <= y`
- `x.__eq__(self, y)`: `x == y`
- `x.__ne__(self, y)`: `x != y`
- `x.__gt__(self, y)`: `x > y`
- `x.__ge__(self, y)`: `x >= y`

- Notes:
  - `x != y` calls `x.__ne__(y)` or negates `x.__eq__(y)`
  - `__lt__` ≈ `!__gt__` and `__le__` ≈ `!__ge__`
  - `__le__` ≉ `__lt__ | __eq__`

Basic customization

- `.__hash__(self)`
  - Called by hash() and used by dict keys and set membership. Must return an integer.

- `.__bool__(self)`
  - Implements truth value testing and called on `bool()`; should return `False` or `True`. Returns `__len__() > 0` if not implemented.

Basic customization

- `.__bytes__(self)`
  - Called by `bytes()` to compute a byte-string representation of an object. This should return a bytes object.

- `.__format__(self, format_spec)`
  - Called by `format()`, f-string evaluation and `str.format()`.

  - E.g. `print(f"{obj:abc}")` will call `obj.__format__('abc')`.
  - See Format Specification Mini-Language for a description of the standard formatting syntax.

example_20_basic_methods.py

- Add a **Book** class with *author* and *title*
- Implements new, init, del, str, repr, eq, bool, hash, format

Emulating Numbers

Emulating Numbers

- `x.__add__(y)`: `x + y`
- `x.__sub__(y)`: `x - y`
- `x.__mul__(y)`: `x * y`
- `x.__matmul__(y)`: `x @ y` (matrix multiplication)
- `x.__truediv__(y)`: `x / y`
- `x.__floordiv__(y)`: `x // y` (returns floor as `int`)
- `x.__mod__(y)`: `x % y`

Emulating Numbers

- `x.__divmod__(y)`: `divmod(x, y)` (returns `(x//y, x%y)`)
- `x.__pow__(y[, modulo])`: `x ** y` **or** `pow(x, y, mod=None)`
- `x.__lshift__(y)`: `x << y`
- `x.__rshift__(y)`: `x >> y`
- `x.__and__(y)`: `x & y` (bitwise and → `01 & 11 = 01`)
- `x.__xor__(y)`: `x ^ y` (bitwise xor → `01 ^ 11 = 10`)
- `x.__or__(y)`: `x | y` (bitwise or → `01 & 11 = 11`)

Emulating Numbers

- `y.__radd__(x)`: `x + y`
- `y.__rsub__(x)`: `x - y`
- `y.__rmul__(x)`: `x * y`
- `y.__rmatmul__(x)`: `x @ y`
- `y.__rtruediv__(x)`: `x / y`
- `y.__rfloordiv__(x)`: `x // y`
- `y.__rmod__(x)`: `x % y`
- `y.__rdivmod__(x)`: `divmod(x, y)`
- `y.__rpow__(x[, modulo])`: `x ** y`, but **not** `pow(x, y, mod=None)`
- `y.__lshift__(x)`: `x << y`
- `y.__rshift__(x)`: `x >> y`
- `y.__and__(x)`: `x & y`
- `y.__xor__(x)`: `x ^ y`
- `y.__or__(x)`: `x | y`

Emulating Numbers

- Prefix the method with “r” for the reflected (swapped/right) operand

- Called if the normal left operand raises an Exception and x and y have different types

- E.g. for `1 + 2.0`:
  - If `int.__add__(1, 2.0)` wasn’t implemented, `float.__radd__(2.0, 1)` would be called

Emulating Numbers

- `x.__iadd__(self, y)`: `x += y`
- `x.__isub__(self, y)`: `x -= y`
- `x.__imul__(self, y)`: `x *= y`
- `x.__imatmul__(self, y)`: `x @= y`
- `x.__itruediv__(self, y)`: `x /= y`
- `x.__ifloordiv__(self, y)`: `x //= y`
- `x.__imod__(self, y)`: `x %= y`
- `x.__ipow__(self, y[, modulo])`: `x **= y`
- `x.__ilshift__(self, y)`: `x <<= y`
- `x.__irshift__(self, y)`: `x >>= y`
- `x.__iand__(self, y)`: `x &= y`
- `x.__ixor__(self, y)`: `x ^= y`
- `x.__ior__(self, y)`: `x |= y`

Emulating Numbers

- Prefix the method with “i” for the augmented assignment operators (e.g. `x += y`).
- Attempts to modify the object in-place and return result
- If undefined, tries `x.__add__(y)` and `y.__radd__(x)`
- E.g. for `1 += 2.0`:
  - Integers are immutable, so `int.__iadd__(2.0)` will return a new int
  - If not implemented, `int.__add__(1, 2.0)` would be called, then `float.__radd__(2.0, 1)`

Emulating Numbers

- `x.__neg__(self)`: `-x`
- `x.__pos__(self)`: `+x`
- `x.__abs__(self)`: `abs(x)`
- `x.__invert__(self)`: `~x`

Emulating Numbers

- `x.__complex__(self)`: `complex(x)`
- `x.__int__(self)`: `int(x)`
- `x.__float__(self)`: `float(x)`
  - These should return the correct type

- `x.__index__(self)`
  - Called when using x as an index, e.g. for slicing, `bin()`, `hex()` and `oct()`. Returns an integer.
  - Calls to `int()`, `float()` and `complex()` fall back to this if corresponding method not defined.

Emulating numbers

- `x.__round__(self[, ndigits])`: `round(x)`

- `x.__trunc__(self)`: `math.trunc(x)`
- `x.__floor__(self)`: `math.floor(x)`
- `x.__ceil__(self)`: `math.ceil(x)`

- All methods should return an `Integral` (e.g. `int`), except `__round__` if `ndigits` present

example_21_numbers.py

- Add a **Dollars** class
- Supports add, radd, iadd

Emulating Containers

- Can be sequences, mappings, or other containers.

Sequences

- Allowable keys:
  - **integers** k, where **0 <= k < N** and N is the length
  - `slice` objects; defines a range of items

- Should implement:
  - `add`, `radd` and `iadd` – concatenate another sequence to end
  - `mul`, `rmul` and `imul` – repeat the sequence n times

- **Mutable** sequences should implement `list` methods:
  - `append`, `count`, `index`, `extend`, `insert`, `pop`, `remove`, `reverse` and `sort`

Mappings

- Recommended to implement `dict` methods:
  - `keys`, `values`, `items`, `get`, `clear`, `setdefault`, `pop`, `popitem`, `copy`, and `update`

- Can inherit from MutableMapping abstract base class

Emulating Containers

- `.__len__(self)`
  - Use `len()` to get the length as an integer >= 0. If `__bool__` not implemented, `bool()` returns `__len__() > 0`.

- `.__length_hint__(self)`
  - Called by operator.length_hint() and returns an estimate of length.

Emulating Containers

- **Note**: `obj[1:2]` is translated to `obj[slice(1, 2, None)]`
- `.__getitem__(self, key)`
  - `self[key]`
  - See docs for what error type to raise.
- `.__setitem__(self, key, value)`
  - `self[key] = value`
- `.__delitem__(self, key)`
  - `del self[key]`

Emulating Containers

- `.__contains__(self, item)`
  - `item in self`, and `item not in self`
  - Returns a boolean.
  - For mappings, check key containment, not values.
  - See docs for fallbacks if `__contains__` is not implemented.

- `.__missing__(self, key)`
  - Called on subclasses of `dict` if `__getitem__` (`self[key]`) is passed a key not in the dictionary.

Emulating Containers

- `.__iter__(self)`
  - Called when an iterator is required for a container.
  - Returns an iterator object (iterator class or method uses **`yield`**).
  - For mappings, it should iterate over the keys.

- `.__reversed__(self)`
  - Called by built-in `reversed()` to implement reverse iteration.
  - Returns an iterator object that iterates in reverse order.
  - If not implemented, falls back to the sequence protocol with `__len__` and `__getitem__`.

example_22_containers.py

- Add a `LinkedList` **sequence**
- Implements append(), len, getitem, setitem, delitem, iter,

Resources – Dunder Methods

- [RealPython] **Python's Magic Methods**
  - https://realpython.com/python-magic-methods/

- [PythonMorsels] **Every dunder method in Python**
  - https://www.pythonmorsels.com/every-dunder-method/

Other

- Iterators
- Attribute access
- Callables
- Context managers
- Buffer types
- Descriptors
- Coroutines

Emulating Iterators

- `.__iter__(self)`
  - Returns self.
- `.__next__(self)`
  - Returns the next item from the iterator.
  - Raise StopIteration if no further items.

- Asynchronous
  - `.__aiter__(self)`
  - `.__anext__(self)`

Resources – Linked Lists and Iterators

- [RealPython] **Linked Lists in Python: An Introduction**
  - https://realpython.com/linked-lists-python/

example_23_iterator.py

- Same as `LinkedList` as ex 21
- Emulates a `LinkedListIterator` and returns it from `LinkedList.__iter__()`

Customizing Attribute Access

- `.__getattribute__(self, name)`
  - Called on `self.name` or `getattr(self, name)`
  - Raise an `AttributeError` if name is not valid
  - To avoid infinite recursion, getting attributes within this method should call the base class, `object.__getattribute__(self, name)`

- `.__getattr__(self, name)`
  - Called if `self.name` raises an `AttributeError` (or called explicitly)

Customizing Attribute Access

- `object.__setattr__(self, name, value)`
  - Called on `self.name = value` instead of saving to `__dict__`
- `object.__delattr__(self, name)`
  - Called on `del self.name`
- `object.__dir__(self)`
  - Called on `dir()` and must return an iterable.
  - `dir()` will convert to a list and sort it.

example_24_attr_access.py

- Class `SharedAttributes`
  - All attributes are saved on a class `dict`

Emulating Callables

- `object.__call__(self[, args...])`
  - Called when the instance is “called” as a function.
  - If defined, `x(arg1, arg2, ...)` roughly translates to `type(x).__call__(x, arg1, ...)`.

```
class Multiply:
    def __init__(self, mult_by):
        self.mult_by = mult_by

    def __call__(self, num):
        return self.mult_by * num
```

```
mult_by_3 = Multiply(3)
mult_by_3(5)  # 15
```

Using `with` Statements

```
with open('filename.txt') as file:
    contents = file.read()
```

- This ensures the file is closed after the `with` block exits.
- Closes the file, even if exceptions were raised within the `with` block.

- Otherwise, use **`try`**/**`finally`**, where **`finally`** closes the file.

```
file = open('filename.txt')
try:
    contents = file.read()
finally:
    file.close()
```

Context Managers (`with` Statements)

- Use a **`with` statement** to wrap the execution of a block of code
- `object.__enter__(self)`
  - Called on `with obj():`
  - Can save return value using `with obj() as ret_val:`
- `object.__exit__(self, exc_type, exc_value, traceback)`
  - Called when exiting **`with`** code block, even if exception thrown.
  - If no exception was thrown, all args are None.
  - Return True to suppress thrown exceptions
- Asynchronous
  - `.__aenter__(self)`
  - `.__aexit__(self, exc_type, exc_value, traceback)`

Using `with` Statements

```
with open('filename.txt') as file:
    contents = file.read()
```

- Calls `__enter__` code at the beginning
- Allows `__exit__` code to be run afterwards, even after an exception

```
file = open('filename.txt')
try:
    contents = file.read()
finally:
    file.close()
```

- With **`try`**/**`finally`**, `__enter__` code is called before **`try`**
- **`finally`** contains `__exit__` code

Using `with` Statements

- Generic example

```
with ContextManager() as value:
    ...
```

- Is roughly equivalent to

```
cm = ContextManager()
value = cm.__enter__()
e = None
try:
    ...
except Exception as exc:
    e = exc
    raise e
finally:
    cm.__exit__(type(e), str(e), e.__traceback__)
```

Resources – Context managers

- [Medium] **Mastering Context Managers in Python**
  - https://python.plainenglish.io/mastering-context-managers-in-python-14ff60d608a8

- [RealPython] **Context Managers and Python's with Statement**
  - https://realpython.com/python-with-statement/

example_25_context_manager.py

- Add a **`BlueConsole`** class

- Inside the context, text prints blue
- Outside the context, text prints normally

Descriptors 

- Descriptor protocol:
  - Allows attributes to have special behavior (e.g. dynamically returned), instead of looking up a value in `__dict__`.
  - Used internally by `classmethod()`, `staticmethod()`, `property()`, and `__slots__`.

- Implementing just `__get__()` creates a non-data descriptor.
- Implementing `__set__()` or `__delete()__` creates a data descriptor.
- Data descriptors have precedence during lookups.

Implementing descriptors

- `.__get__(self, instance, owner=None)`
  - Returns the computed attribute value or `AttributeError`.
  - **`instance`** is the instance object it was called on (or None)
  - **`owner`** is the class object it was called on
- `.__set__(self, instance, value)`
  - Set the attribute on **`instance`** to a new value.
- `.__delete__(self, instance)`
  - Delete the attribute on **`instance`**.
- `. __objclass__`
  - Specifies the class where this object was defined.

Simple descriptor example

```
import os
class DirectorySize:
    def __get__(self, obj, objtype=None):
        return len(os.listdir(obj.dirname))

class Directory:
    size = DirectorySize() # Descriptor instance

    def __init__(self, dirname):
        self.dirname = dirname # Regular instance attribute
```

```
>>> g = Directory('games')
>>> s.size
```

Resources – Descriptors

- [Python] **Descriptor Guide**
  - https://docs.python.org/3/howto/descriptor.html

- [RealPython] **Python Descriptors: An Introduction**
  - https://realpython.com/python-descriptors/

example_26_descriptors.py

Emulating Buffer Types

- Buffer protocol:
  - Allows objects to expose efficient access to a low-level memory array.

- Used by `bytes`, `bytearray`, `memoryview` and `array.array`.
- Third-party libraries may use it for image processing or numeric analysis.

- Usually written in C, but can be emulated in Python with `__buffer__()` and `__release_buffer__()`.
  - New in Python 3.12 – See PEP 688

Emulating Buffer Types

- `object.__buffer__(self, flags)`
  - Called when a buffer is requested from self.
  - **`flags`** is an integer for the kind of buffer requested, e.g. read-only
  - Must return a `memoryview` object.

- `object.__release_buffer__(self, buffer)`
  - Called when a buffer is no longer needed.
  - **`buffer`** is the `memoryview` object returned by `__buffer__()`.
  - Release any resources associated with the buffer.
  - Must return None.

Emulating Generic Types

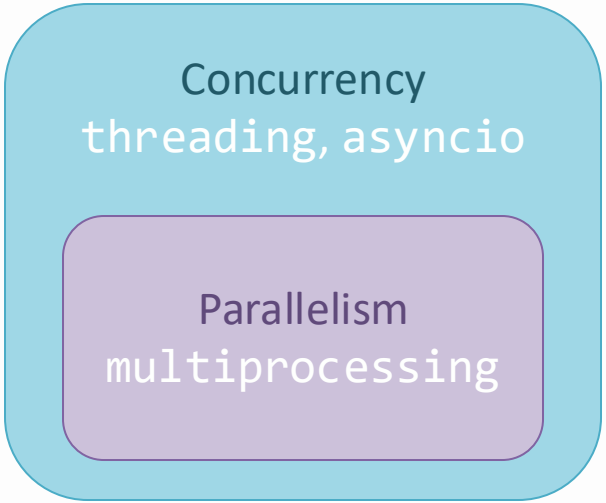
- `.__class_getitem__(cls, key)`
  - Called on `cls[key]`.
  - **Remember**: `instance[key]` calls `instance.__getitem__(key)`
  - Inherit from a class that implements this, like `typing.Generic`

- Allows parameterization of a generic types when using type hinting

```
from typing import Generic, TypeVar
T = TypeVar('T')

class MyClass(Generic[T]):
    def __init__(self, value: T):
        ...

a: MyClass[int] = MyClass(1)
```

Concurrency

- **Concurrency**
  - Managing lots of things at once.
  - E.g. when making HTTP requests, do other work while awaiting a response.
- **Parallelism**
  - Doing lots of things at once.
  - E.g. run multiple scripts at the same time to make use of multiple CPU cores.



Coroutines and Generators

- ***Coroutines*** and ***generators*** are functions that can suspend themselves and resume later, allowing other code to run.
- ***Generators*** are functions that use the **`yield`** keyword to return a value to the calling context and continue executing later.
- ***Coroutines*** can either be:
  - A *generator* that, after yielding a value, also accepts new values from the calling context (*classic coroutines*).
  - A *function* defined with **`async def`**. It may contains an **`await`** keyword, which will suspend the function until the awaiting task is complete (*native coroutines*).
- See Fluent Python Chapter 19-21

Awaitable Objects

- Coroutines and generators are awaitable (they can pause execution of a function while waiting for a result).

- `.__await__(self)`
  - Must return an iterator.
  - Should be used to implement awaitable objects.
  - For instance, asyncio.Future implements this method to be compatible with the **`await`** expression.

Asynchronous Objects

- *Asynchronous iterators* – used in **`async for`** statements
  - **`.__aiter__(self)`**
    - Must return an asynchronous iterator object.
  - **`.__anext__(self)`**
    - Must return an awaitable resulting in a next value of the iterator. Should raise a StopAsyncIteration error when the iteration is over.

- *Asynchronous context managers* – used in **`async with`** statements
  - **`.__aenter__(self)`**
    - Similar to `__enter__()`, but must return an awaitable.
  - **`.__aexit__(self, exc_type, exc_value, traceback)`**
    - Similar to `__exit__()`, but must return an awaitable.

example_27_async.py

Resources – Async

- [Medium] **An Intro to Asynchronous Programming in Python**
  - https://medium.com/velotio-perspectives/an-introduction-to-asynchronous-programming-in-python-af0189a88bbb
- [RealPython] **Getting Started With Async Features in Python**
  - https://realpython.com/python-async-features/
- [RealPython] **Async IO in Python: A Complete Walkthrough**
  - https://realpython.com/async-io-python/

Metaprogramming

Metaprogramming

- Allow program to modify or generate code at runtime
  - Write programs that manipulate programs
- Used in frameworks, libraries, template engines, and for code generation
- Mechanisms for metaprogramming in Python:
  - `getattr()`, `setattr()`
  - Decorators – Practical decorators talk
  - Dynamic attributes and properties
  - **Custom class dunder methods**
  - **Metaclasses**

Custom class creation

- `.__init_subclass__(cls)`
  - Called on a parent class when another class inherits from it.

```
class BaseClass:
    def __init_subclass__(cls):
        print('hi')

class SubClass(BaseClass): # prints 'hi'
    pass
```

Custom class creation

- `.__set_name__(self, owner, name)`
  - If this class is set as a class attribute on another class, this is called at the time the owning class owner is created.

- For code:

```
class A:
    x = C()
```

  - `x.__set_name__(owner=A, name='x')` is called, or
  - `C.__set_name__(x, owner=A, name='x')`

Metaclasses

- Allows for creation of customized classes at runtime.
- Frequently used in libraries and frameworks that require a user to subclass a base class.
- Helpful for understanding framework code, but 99% of developers don't need to write it.

```
from enum import Enum

class Rating(Enum):
    good = 1
    okay = 0
    bad = -1
```

```
print(Rating.good)          # Rating.good
print(Rating.good.name)     # good
print(Rating.good.value)    # 1
```

https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Metaprogramming.html

Metaclasses

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

- When a class definition is executed, the following steps occur:
  - MRO entries are resolved;
  - the appropriate metaclass is determined;
  - the class namespace is prepared;
  - the class body is executed;
  - the class object is created.

Metaprogramming

- `.__new__(cls, name, bases, attrs)`
  - `name` – name of the class
  - `bases` – tuple of base classes
  - `attrs` – the namespace dictionary

```
class Meta(type):  # inherit from type
    def __new__(cls, name, bases, attrs):
        attrs['x'] = 100
        return super().__new__(cls, name, bases, attrs)
```

```
class MyClass(metaclass=Meta):  # set metaclass
    ...
```

```
print(MyClass.x)  # prints 100
```

Metaprogramming

- `object.__mro_entries__(self, bases)`
  - If a base is not an instance of `type`, then an `__mro_entries__()` method is searched on the base.
  - If found, the base is substituted with the result of a call to `__mro_entries__()` when creating the class.
  - The method is called with the original bases tuple passed to the bases parameter.
  - Must return a tuple of classes that will be used instead of the base.

- `metaclass.__prepare__(name, bases, **kwds)`
  - Called during class pre-construction.
  - Returns a class namespace (i.e. `__dict__`) that gets passed into `__new__()`.

Customizing type checks 

- These are only called on metaclasses of an instance

- `.__instancecheck__(self, instance)`
  - Called by `isinstance(instance, class)`
  - Return `True` if instance should be considered a (direct or indirect) instance of class.

- `.__subclasscheck__(self, subclass)`
  - Called by `issubclass(subclass, class)`
  - Return True if subclass should be considered a (direct or indirect) subclass of class.

[Metaclasses] are deeper magic than 99% of users should ever worry about.

If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).

- Tim Peters, prolific Python contributor

example_28_metaprogramming.py

Resources – Meta programming

- [PEP 3115] **Metaclasses in Python 3000**
  - https://peps.python.org/pep-3115/
- [RealPython] **Python Metaclasses**
  - https://realpython.com/python-metaclasses/
- [Fluent Python] **Class Metaprogramming**
  - https://learning.oreilly.com/library/view/fluent-python-2nd/9781492056348/ch24.htm

Conclusion

Resources – Books

- [Dan Bader] **Python Tricks**
  - https://realpython.com/products/python-tricks-book/
- [Brett Slatkin] **Effective Python**
  - https://learning.oreilly.com/library/view/effective-python-90/9780134854717/
- [Luciano Ramalho] **Fluent Python, 2nd Edition**
  - https://learning.oreilly.com/library/view/fluent-python-2nd/9781492056348/
- [Michal Jaworski, Tarek Ziadé] **Expert Python Programming**
  - https://learning.oreilly.com/library/view/expert-python-programming/9781801071109/

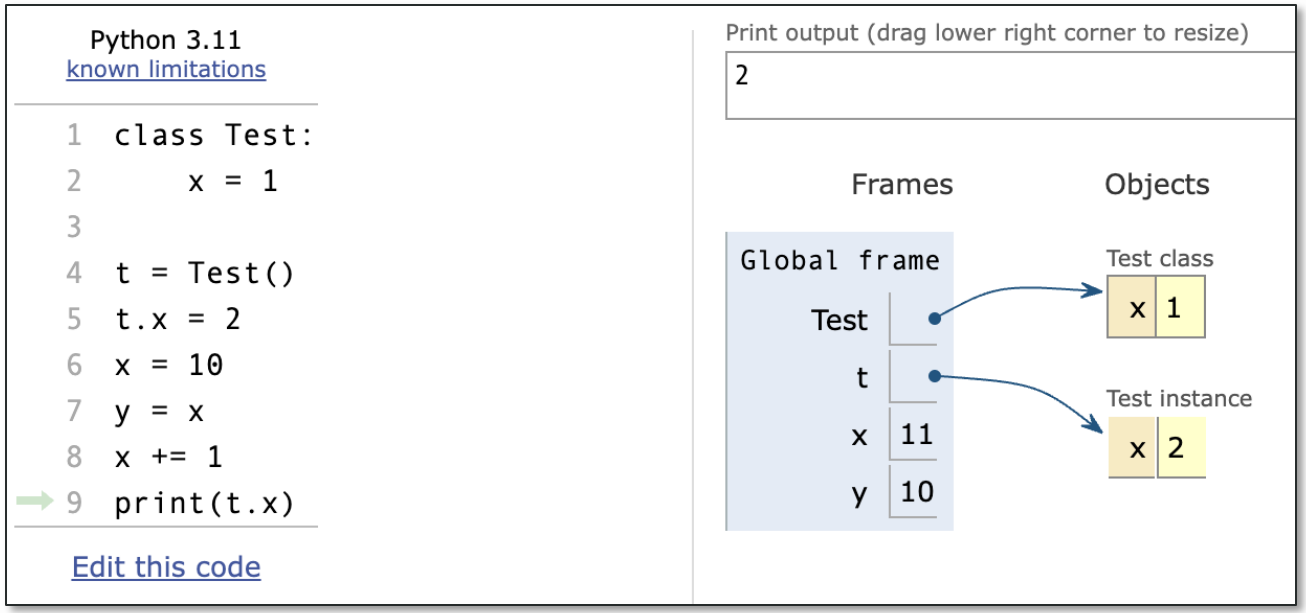
Resources – Websites

- [Yaniv Aknin] **Python Innards**
  - https://tech.blog.aknin.name/tag/internals/
  - Interesting, in-depth blog posts about Python internals
  - **Warning**: some parts are out of date

- [Trey Hunner] **Python Morsels**
  - https://www.pythonmorsels.com/
  - Python exercises for more advanced concepts

Resources – Websites

- [Python Tutor] **Online Compiler and Visual Debugger for Python**
  - https://pythontutor.com/
  - Step through code and see memory and object content over time



Resources – Great videos

- **Facts and Myths about Python names and values - PyCon 2015**
  - https://www.youtube.com/watch?v=_AEJHKGk9ns
  - *Ned Batchelder*

- **What Does It Take To Be An Expert At Python?**
  - https://www.youtube.com/watch?v=7lmCu8wz8ro
  - *James Powell*

Resources – More great videos

- **Beyond PEP 8 -- Best practices for beautiful intelligible code**
  - https://www.youtube.com/watch?v=wf-BqAjZb8M&ab_channel=PyCon2015
  - *Raymond Hettinger*

- **Stupid things I've done with Python**
  - https://www.youtube.com/watch?v=jIM5urFHf2k&ab_channel=PythonIreland
  - *Mark Smith*

- **Elegant Solutions For Everyday Python Problems**
  - https://www.youtube.com/watch?v=WiQqqB9MlkA&ab_channel=PyCon2018
  - *Nina Zakharenko*

Beginner Live Trainings by Arianne

- **Introduction to Python Programming**
  - Variables, functions, conditionals, lists, loops
  - Skill level – 1/10
- **Programming with Python: Beyond the Basics**
  - Dictionaries, exceptions, files, HTTP requests, web scraping
  - Skill level – 2/10
- **Python Environments and Best Practices**
  - Virtual envs, testing, debugging, PyCharm tips, git, modules
  - Skill level – 3/10
- **Hands-on Python Foundations in 3 Weeks**
  - Multi-week course that covers most of the above material
  - Skill level 1-3

Intermediate Live Trainings by Arianne

- **Object-Oriented Programming in Python**
  - Classes, dunder methods, and decorators
  - Skill level – 3/10

- **Python Data Structures and Comprehensions**
  - Overview of data structures from the standard library, Numpy and Pandas
  - Skill level – 4/10

- **Python Under the Hood**
  - CPython overview, dunder variables and methods, inspecting objects, debugging
  - Skill level – 5/10

- **Learn GraphQL in 4 Hours**
  - Explore GraphQL features
  - Build a GraphQL API in Django and Node.js
  - Skill level – 5/10

Video Trainings by Arianne

- **Introduction to Python LiveLessons - Link**
  - Very beginner content w/ brief intro to data analysis and web development
- **Next Level Python LiveLessons - Link**
  - Material from this class
  - Setting up Python projects with virtual environments and git
  - Testing, debugging, and understanding modules
- **Introduction to Django - Link**
  - Understand basics of creating web applications in Django
  - Start a new project and app
  - Overview of different components and features
- **Rethinking REST: A hands-on guide to GraphQL and Queryable APIs - Link**
  - Explore GraphQL features
  - Build a GraphQL client in JavaScript and a server in Django or Node.js

Interactive labs by Arianne

Hands-On Python Foundations course

1. Getting Started with Python
2. Types, Variables and Strings
3. Functions and Control Flow
4. Data Structures
5. Exceptions and File Handling
6. Requests and APIs
7. Virtual Environments and Pip
8. Intro to Classes
9. Modules and Packages

Thanks!

Questions?

Email me at

- arianne.dee.studios@gmail.com