SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 19

Fabian Wermelinger
Harvard University
CS107 / AC207

Thursday, November 3rd 2022

LAST TIME

- Continuation with binary search trees
- Binary tree traversal
- Other common data structures
- Priority queues and heaps

TODAY

Main topics: ***Python generators and coroutines***

Details:

- Generators
- Coroutines
- Python internals: objects, bytecode and interpreter

GENERATORS

- In lecture 17 we discussed the ***iterator*** design pattern. Iterators are used for the following:

> Provide a way to access the elements of an aggregate object *sequentially without exposing its underlying representation*.

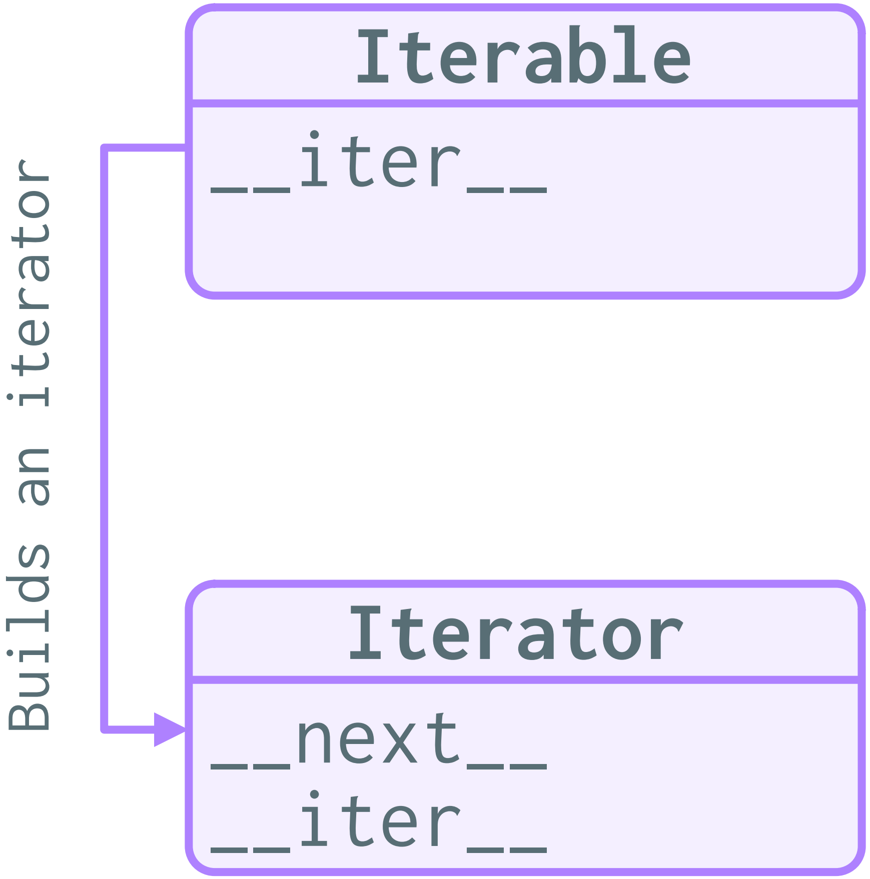
- Iterators are *fundamental* for processing of data. So far our conception was that the *data* we are iterating over lives in Random Access Memory (RAM) on your computer (e.g. `list`, `tuple` or a linked lists).
- What if the data we need to process is *too large* to fit into the RAM? Then we need a way to ***lazily*** fetch new elements as we call `next()`.

> Every ***generator*** *is an iterator* and fully implements the iterator interface. An *iterator* retrieves its items from a collection (e.g. a `list`), while a *generator* can ***produce*** items upon request (it is "lazy").

→ the terms *"iterator"* and *"generator"* are often used interchangeably. Be aware of the difference above!

GENERATORS

Recall: the iterator interface (Python obtains *iterators* from *iterables*)



- ***Iterable:*** an object `x` that is *iterable* implements the `__iter__` special method, which returns *a new* iterator for every `iter(x)` call.
- ***Iterator:*** an iterator implements the standard interface:
  - `__next__`: returns the next available item. Raises `StopIteration` when there are no more items.
  - `__iter__`: returns `self`. (Allows iterators to be used where an *iterable* is expected.)
- ***Generator:*** has the same interface as an iterator.

GENERATORS

- ***Generators*** were added in Python 2.2 in 2001 and introduced the new keyword → `yield`.
- Generators are defined in PEP 255 -- Simple Generators.
- A generator ***function*** is essentially the same as a regular function, except that it *yields* (or produces a value) rather than *returning* from a function (*a generator function can return as well, however*).
- A generator "`g`" *yields a value whenever we call `next(g)`* on it. It is then *temporarily suspended* until we call `next(g)` again.
→ generator functions are *lazy* and execute *on demand*.
- A generator function raises `StopIteration` if either a `return` statement or the end of the generator function body is reached.

GENERATOR FUNCTION

- Any Python function that has a `yield` keyword in its body is a *generator function*.
- A generator function results in a *generator object* when called.
- You can think of generator functions as *factories* for generator objects.

Example:

```python
def gen_107():
    """Generator function"""
    yield 1
    yield 0
    yield 7
```

- The generator yields the value that follows every `yield` keyword.
- The `next()` built-in advances the generator the same way it does for iterators. *It will advance to the next `yield` keyword.*

```python
>>> from inspect import getgeneratorstate
>>> gen_107
<function gen_107 at 0x7fe79f4e7310>
>>> g = gen_107()
<generator object gen_107 at 0x7fe79f57d820>
>>> getgeneratorstate(g)
'GEN_CREATED'
>>> next(g)
1
>>> getgeneratorstate(g)
'GEN_SUSPENDED'
>>> list(g)
[0, 7]
>>> getgeneratorstate(g)
'GEN_CLOSED'
```

GENERATOR FUNCTION

Example:

```python
def gen_107():
    """Generator function"""
    yield 1
    yield 0
    yield 7
```

```python
>>> for i in gen_107():
...     i
...
1
0
7
```

- The generator yields the value that follows every `yield` keyword.
- The `next()` built-in advances the generator similarly to iterators.

→ a generator implements the standard iterator interface. *A generator can be used interchangeably with iterators!*

Generator states: (see `inspect.getgeneratorstate`)

State	Description
`GEN_CREATED`	Waiting to start execution
`GEN_RUNNING`	Currently being executed by the interpreter
`GEN_SUSPENDED`	Currently suspended at a `yield` expression
`GEN_CLOSED`	Execution has completed

GENERATOR FUNCTION

Example:

```python
def gen_107():
    """Generator function"""
    yield 1
    yield 0
    return
    yield 7 # never executed!
```

```python
>>> for i in gen_107():
...     i
...
1
0
```

You can use the `return` keyword inside a generator function. Upon `return`, the generator will raise `StopIteration` and `yield 7` will *never be reached.*

A generator function creates separate instances of *generator objects* → a generator function is a *factory* for generator objects:

```python
>>> a, b = gen_107(), gen_107()
>>> a; b
<generator object gen_107 at 0x7fe79f2cf120>
<generator object gen_107 at 0x7fe79f2cf190>
>>> iter(a); iter(b)
<generator object gen_107 at 0x7fe79f2cf120>
<generator object gen_107 at 0x7fe79f2cf190>
```

LINKED LIST ITERATOR REVISITED

Iterator implementation for linked list in lecture 17:

```
class LinkedList:
    # other code skipped

    def __iter__(self):
        return LinkedListForwardIterator(
            self.first)

class LinkedListForwardIterator:
    def __init__(self, start):
        self.node = start

    def __next__(self):
        if self.node != None:
            curr_node = self.node
            self.node = self.node.next
            return curr_node
        else:
            raise StopIteration

    def __iter__(self):
        return self
```

Same functionality implemented with a generator function:

```
class LinkedList:
    # other code skipped

    def __iter__(self):
        node = self.first
        while node != None:
            yield node
            node = node.next
```

- `__iter__` is now a generator function *(a factory of generators)*.
- A generator implements the standard iterator interface, otherwise this would not work!
- ***No need to write an extra class for it! Less code → way more elegant.***

GENERATOR EXPRESSIONS

Comprehensions:

- Comprehensions provide a concise way to build lists, sets or dictionaries.
- `list` comprehension example:

```
>>> [x for x in range(5)]
[0, 1, 2, 3, 4]
```

- `set` comprehension example:

```
>>> {x for x in [0, 0, 1, 2, 2]}
{0, 1, 2}
```

- `dict` comprehension example:

```
>>> {k:v for k, v in [(0,'a'), (1,'b'), (2,'c')]}
{0: 'a', 1: 'b', 2: 'c'}
```

- Comprehensions are built ***eagerly***. Once created, the complete data structure exists in memory (RAM).

Generator expressions:

- A generator expression can be thought of as a ***lazy*** version of a list comprehension.
- It ***does not*** eagerly build a list, but returns a generator that will *lazily produce the items on demand*.
- A generator expression is written using parentheses `()` instead of brackets `[]` or curly braces `{}`.
- Generator expression example:

```
>>> (x.upper() for x in list('abc'))
<generator object <genexpr> at 0x7f6c2d732430>
```

→ you cannot use the `yield` or `yield from` keywords in a generator expression!

GENERATOR EXPRESSIONS

Iterable via list comprehension:

```python
# generator function
def gen():
    print('Start')
    yield 'A'
    print('Continue')
    yield 'B'
    print('End')

# list comprehension (eager)
lc = [x for x in gen()]

# iterate over list
print('Enter loop')
for i in lc:
    print(i)
```

Output (eager):

```
Start
Continue
End
Enter loop
A
B
```

Iterable via generator expression:

```python
# generator function
def gen():
    print('Start')
    yield 'A'
    print('Continue')
    yield 'B'
    print('End')

# generator expression (lazy)
ge = (x for x in gen())

# iterate over generator expression
print('Enter loop')
for i in ge:
    print(i)
```

Output (lazy):

```
Enter loop
Start
A
Continue
B
End
```

GENERATOR EXPRESSIONS

- Generator expressions are syntactic sugar in Python.
- Proposal for generator expressions → PEP 289
- They are nice in places where you want to be *brief and concise* (same with comprehensions).
- *Generator functions* and *generator expressions* both return *generator objects* → *they are both just generator factories.*
- *Generator functions* are much more flexible! They allow for *multiple statements* and more complex code.
- *Generator functions* can further be used as *coroutines*, a generalization that will be introduced later.

GENERATOR FUNCTIONS IN THE STANDARD LIBRARY

- The Python standard library has many *generator utilities implemented* → https://docs.python.org/3/library/index.html
- You should be aware of some commonly used tools in order *not to reinvent the wheel*.
- Useful categories for data transformations include:
  - Filters
  - Maps
  - Merge of inputs
  - Expansion of input into multiple outputs
  - Rearrangements
  - Reductions
- Most of the tools above are available in the `itertools` module. Others are built-in without need of importing modules.

GENERATOR FUNCTIONS IN THE STANDARD LIBRARY

Example: `os.walk`

Example directory structure:

```
.
├── file_top
├── oswalk.py
├── subdir1
│   ├── file_subdir1
│   └── subsubdir1
│       └── file_subsubdir1
└── subdir2
    └── file_subdir2
```

`os.walk` generator in Python:

```
>>> import os
>>> for path, dirs, files in os.walk('./'):
...     print(f'{path}\n\tdirs:  {dirs}\n\tfiles: {files}')
...
./
        dirs:  ['subdir1', 'subdir2']
        files: ['oswalk.py', 'file_top']
./subdir1
        dirs:  ['subsubdir1']
        files: ['file_subdir1']
./subdir1/subsubdir1
        dirs:  []
        files: ['file_subsubdir1']
./subdir2
        dirs:  []
        files: ['file_subdir2']
```

- `os.walk` is very powerful for exploring data in file systems.
- `os.walk` returns a generator object that yields the current directory `path` along with lists of directories `dirs` and `files` contained within it.

GENERATOR FUNCTIONS IN THE STANDARD LIBRARY

Example: `filter`

- Filter elements in an iterable for which a *predicate function* returns true:

```
filter(predicate_function, iterable)
```

- The return object of a call to `filter` is an iterator.
- ***Example:*** filter vowels

```
>>> list(filter(lambda x: x.lower() in 'aeiou', 'What you seek is seeking you.'))
['a', 'o', 'u', 'e', 'e', 'i', 'e', 'e', 'i', 'o', 'u']
```

or types:

```
>>> list(filter(lambda x: isinstance(x, int), [0, 0x1, 0o2, 3.0]))
[0, 1, 2]
```

- The inverse of `filter` is provided by `itertools.filterfalse`:

```
>>> from itertools import filterfalse
>>> list(filterfalse(lambda x: isinstance(x, int), [0, 0x1, 0o2, 3.0]))
[3.0]
```

GENERATOR FUNCTIONS IN THE STANDARD LIBRARY

Example: `map`

- Applies a function to every item of an iterable and *yields* the result.
- You can pass n iterables. In this case the function must take n arguments.
- See `itertools.starmap` for an alternative version to unpack function arguments instead.
- ***Example:***

```
>>> list(map(lambda a, b: (a, b), range(11), list('ABC')))
[(0, 'A'), (1, 'B'), (2, 'C')]  # list('ABC') is the shorter iterable!
```

→ note that `map` stops when the shortest input iterator is exhausted!

- There are many more useful generator functions in `itertools`. Be sure to check them out.

APPLICATION EXAMPLE: LAZY READ OF LARGE DATA

- Generators are useful to create *data readers* for working with data sets.
- These readers *expose the iterator pattern*.
- If your input data set is *very large* it may not fit into RAM entirely. In that case you must produce the data *on the fly* → *lazy read* the data from the disk or tape. *A generator is the right tool for this.*
- ***Example:*** assume you are working with the following function to process data. The input data may either come from an ASCII text file or a NumPy binary file *(e.g. due to different data collection procedures)*:

```python
from itertools import chain
def process(*data_iterables): # can pass multiple input generators...
    val = 0
    item_count = 0
    for item in chain.from_iterable(data_iterables): # ...and chain them together
        val += item  # in this example we just sum up the input data values
        item_count += 1
        if item_count % 10000 == 0: # print progress
            print(f'{item_count} items processed from input')
    return val
```

APPLICATION EXAMPLE: LAZY READ OF LARGE DATA

- ***Example:*** data processing function

```python
from itertools import chain
def process(*data_iterables): # can pass multiple input generators...
    val = 0
    item_count = 0
    for item in chain.from_iterable(data_iterables): # ...and chain them together
        val += item  # in this example we just sum up the input data values
        item_count += 1
        if item_count % 10000 == 0: # print progress
            print(f'{item_count} items processed from input')
    return val
```

- The function takes a number of *data generators* (or just iterables) and chains them together to iterate over all data items combined. ***Examples for data items:*** movie frames, pressure fields from simulations, audio samples, sentences, *anything really*.
- Assume the data generators above yield just one number at a time.
- What if the data file(s) are too large for your physical RAM or if you read from a continuous stream? You must use a *generator* in this case.

APPLICATION EXAMPLE: LAZY READ OF LARGE DATA

- Eager reader for binary data specific to some application *(all data is loaded into RAM)*:

```python
def read_binary_eager(fname): # returns iterable
    with open(fname, 'r') as f: # open file read-only
        return np.fromfile(f, dtype=float)
```

- Lazy reader for binary data specific to some application:

```python
def read_binary_lazy(fname): # returns generator object
    with open(fname, 'r') as f: # open file read-only
        while item := np.fromfile(f, dtype=float, count=1):
            yield item
```

(***Note:*** ":=" is called an *assignment expression* and only works for Python 3.8 and later. See https://peps.python.org/pep-0572/)

- When you use this code in practice you notice that the lazy reader (generator) is ***100x slower***! ***Do you have an idea what could be the problem?***

APPLICATION EXAMPLE: LAZY READ OF LARGE DATA

- Lazy reader for binary data specific to some application:

```python
def read_binary_lazy(fname): # returns generator object
    with open(fname, 'r') as f: # open file read-only
        while item := np.fromfile(f, dtype=float, count=1):
            yield item
```

- *Bandwidth* on a computer is very precious and it is *optimized* for reading more than ***just a few bytes at a time!*** → in the example reader above *only 8 bytes are read at a time* (one 64-bit float).

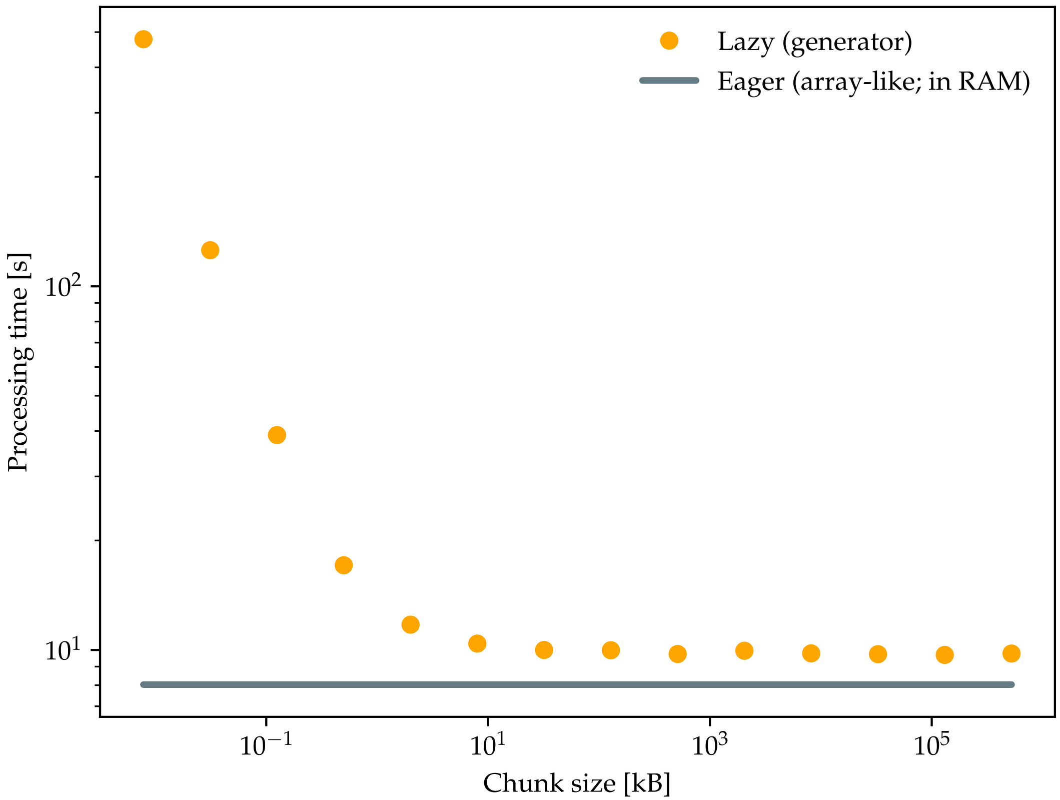
- ***Read at least a few kilo-bytes at a time:***

```python
def read_binary_lazy(fname, chunk_size=1): # returns generator object
    with open(fname, 'r') as f: # open file read-only
        while (chunk := np.fromfile(f, dtype=float, count=chunk_size)).size > 0:
            for item in chunk:
                yield item
```

→ ***this will saturate the memory bandwidth and improve performance!***

APPLICATION EXAMPLE: LAZY READ OF LARGE DATA

Benchmark for lazy load of 512MB file with different chunk size:



- If you read very small data lazily, you will *not saturate the memory bandwidth* and process up to 100x slower!
- You should read ***at least*** *10–100 kilobyte* at a time. This may depend on the architecture you are running on. The larger the chunk size the better.
- Prefer to eager load the full data set if you can afford it.

COROUTINES

- You are already familiar with the concept of a ***function***. They are also called ***subroutines*** (especially in Fortran):
  - Functions allow you to avoid *code duplication*.
  - They form a logical segmentation of the problem. Also enable easier debugging.
  - When you call a function, temporary variables are allocated (called automatic variables) that exist during the lifetime of the function only. Examples are function arguments or local variables in the function body.
  - A function has one entry and may have multiple return points. It is *asymmetric*. Once you return, the memory for the temporary variables is released.
- A function is a special case of a more general concept called a *coroutine*.
  - Functions are *asymmetric* between caller and callee.
  - Coroutines are *symmetric* between caller and callee → *you can enter and leave a coroutine many times*.

You may think of two coroutines as a "*team*" of programs that repetitively call each other with different input each time.

COROUTINES

- ***A coroutine is syntactically the same as a generator:*** a function with a `yield` keyword in its body.
- In a coroutine the `yield` keyword usually appears *on the right side of an expression.* ***Example:***

```
def coroutine():
    while True:
        x = yield # yield may obtain something and we assign it to x!
        print(x)
```

  - ***Note:*** `yield` *may or may not* produce a value! If there is no expression after `yield` → it will yield None *(like in the example code above).*
  - Unlike a generator, a coroutine ***can receive data from the caller*** by calling `c.send(data)` (`c.send(data)` returns what `next(c)` would return):

```
>>> c = coroutine()
>>> next(c) # we must prime the coroutine by calling next() before use
>>> y = c.send('Hello CS107/AC207')
Hello CS107/AC207
>>> print(y)
None
```

COROUTINE EXAMPLE

```python
1  from inspect import getgeneratorstate
2
3  def coroutine():
4      ncalls = 0
5      while True:
6          x = yield ncalls
7          ncalls += 1
8          print(f'coroutine(): {x} (call id: {ncalls})')
9
10 def main():
11     c = coroutine()
12     print(getgeneratorstate(c))
13     next(c)  # prime the coroutine; next() returns 0 here
14     print(getgeneratorstate(c))
15
16     c.send('CS107')
17     print('main(): control back in main function')
18     last_call = c.send('AC207')
19     print(f'main(): called coroutine() {last_call} times')
20
21     c.close()
22     print(getgeneratorstate(c))
23
24 if __name__ == "__main__":
25     main()
```

```
GEN_CREATED
GEN_SUSPENDED
coroutine(): CS107 (call id: 1)
main(): control back in main function
coroutine(): AC207 (call id: 2)
main(): called coroutine() 2 times
GEN_CLOSED
```

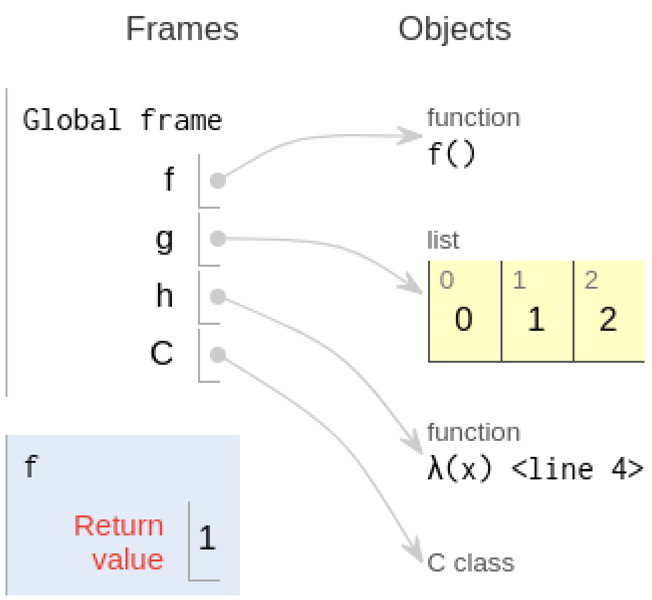
- `line 11` creates the coroutine (nothing has been run yet).
- `line 13` primes the coroutine. This means activate it and run until the first `yield`, then suspend it.
- `line 16` sends data to the suspended coroutine. This will activate it and run until the next `yield` is reached. The `.send()` call is similar to `next()` except that we also send data.
- Coroutines can be shutdown using the `.close()` method in case it never reaches a `return` statement (or end of function).

GENERATORS/COROUTINES RECAP

- A generator can pause at a `yield` statement. It yields a value back to the caller. The generator state is suspended until `next()` is called on it again.
  → think of `yield` as control flow.
- A generator implements the standard iterator interface.
- Coroutines are an advanced programming concept that involves entering and returning from functions which are in an *intermediate* execution state. A Python coroutine *extends* a generator with `.send`, `.close` and `.throw` methods. *Coroutines and generators are conceptually very different*.
- Guido van Rossum: there are three different styles of code you can write using generators:
  1. The traditional "*pull*" style (***example:*** linked list `__iter__` we discussed, see PEP 255 and PEP 380).
  2. The "*push*" style (i.e., the coroutines that we discussed here using `.send` to push, see PEP 342).
  3. Concurrent tasks (coroutines with `async` and `await` syntax for concurrent programming, see PEP 492). We do not discuss concurrent programming in this class.

PYTHON INTERNALS: OBJECTS AND FRAMES

Recall: the sketches we saw during the pythontutor examples



- ***Frames:*** "frame objects" that execute code (imagine a *stack* data structure: the blue shaded frame is at the top of the stack).
- ***Objects:*** any other Python objects. Functions, classes, data structures, etc.

- Frame objects *execute* code and form a *sequence* in a *stack* data structure.
- Arrows indicate *references to objects in memory*.
- When we enter a function `f()`, a new *frame* is *pushed* onto the stack and executes.
- When done, the function frame is *popped* off the stack and we return to the caller frame (global frame here).
- The data structure used to organize frames is a LIFO stack. ***Will that work for coroutines?***

PYTHON INTERNALS: OBJECTS

All the data stored in a Python program is built around the concept of an object.

Terminology:

- Every piece of data is stored in an *object*. *This includes Python frames and code.*
- Each object has an *identity*, a *type* (also known as its class) and a *value*.
- The identity of an object is its location in memory. *Names* store a *reference* to that a specific memory location.
- The *type* of an object describes the internal representation as well as methods and operations it supports → implemented in a `class`.
- When an object of a specific type is created, we called it an *instance* of that type. *After an instance is created, its identity and type can no longer be changed.*
- If an object's value can be modified, we call it *mutable*, otherwise it is said to be *immutable*.
- *Containers* or *collections* are objects that contain references to other objects.
- Because everything in Python is represented by objects, they are said to be *first class*.

PYTHON INTERNALS: OBJECTS

Example: user defined function object

- User defined functions are *callable* objects created at the module level by using `def` or `lambda`. Functions are ***first class*** objects in Python.

```
>>> def f():
...     pass
...
>>> g = lambda x: x
>>> f.__code__; g.__code__
<code object f at 0x7fd88a3fac90, file "<stdin>", line 1>
<code object <lambda> at 0x7fd88a3fabe0, file "<stdin>", line 1>
```

- A user-defined function `f` has the following attributes:

Attribute	Description
`f.__doc__`	Documentation string
`f.__name__`	Function name
`f.__dict__`	Dictionary containing function attributes
`f.__code__`	Byte-compiled code
`f.__defaults__`	Tuple containing the default arguments
`f.__globals__`	Dictionary defining the global namespace
`f.__closure__`	Tuple containing data related to nested scopes

- Python code ***is compiled*** into *bytecode* objects *on the fly*.
- Python is an *interpreted* language. Under the hood, however, code is transformed into bytecode objects. The interpreter is a virtual machine.
- Running your code for the first time is slower due to bytecode generation. The result is *cached* in `.pyc` files for faster subsequent execution.

RECAP

- Generators
- Coroutines
- Python internals: objects, bytecode and interpreter

Further reading:

- Generator and yield expressions: https://docs.python.org/3/reference/expressions.html#generator-expressions
- Chapters 14 and 16 in Luciano Ramalho, *"Fluent Python: Clear, Concise, and Effective Programming"*, O'Reilly Media, 2015