CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 18

Thursday, November 4th 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

- Binary trees and principal binary tree traversal
- Priority queues and heaps

OUTLINE

- Generators
- Coroutines
- `python` internals: objects, bytecode and interpreter

GENERATORS

- In a previous lecture we discussed the *iterator* design pattern. Its intent was the following:

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

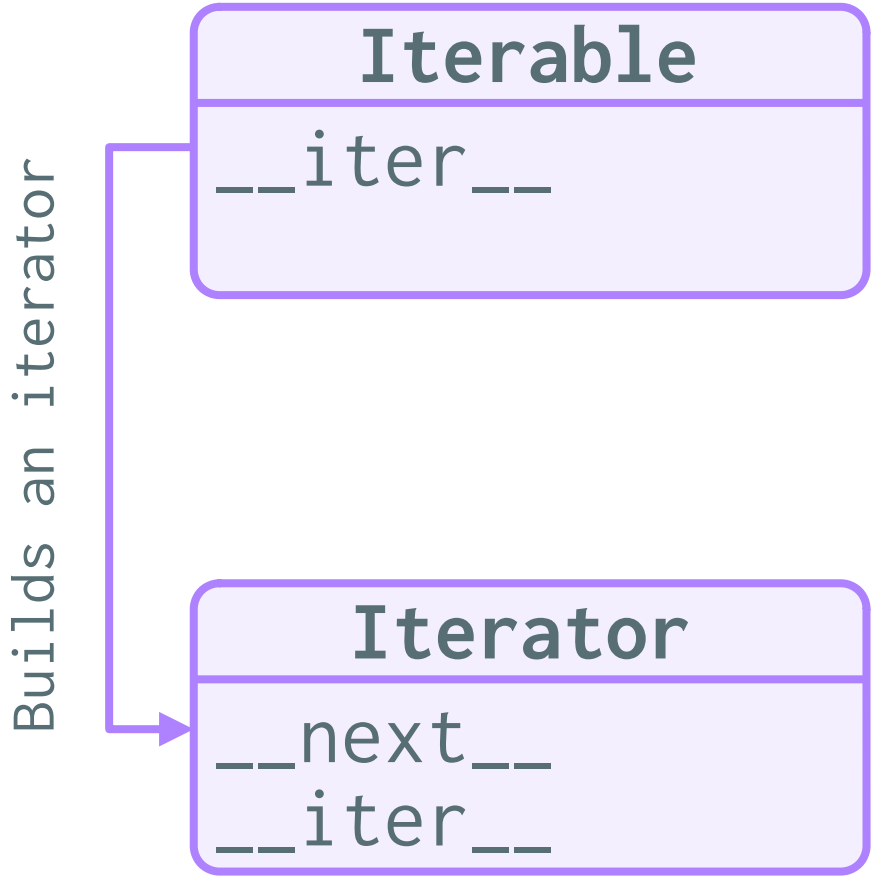
- Iterators are *fundamental* for processing data. So far our conception was that the *data* we are iterating over lives in the Random Access Memory (RAM) of your computer (e.g. `list`, `tuple` or a linked lists).
- What if the data we need to process is *too large* to fit into the RAM? Then we need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

A ***generator*** has the same interface as an iterator.

GENERATORS

- ***Generators*** were added in `python 2.2` (2001) and introduced a new keyword: `yield`.
- They are defined in PEP 255 -- Simple Generators.
- A generator *function* is essentially the same as regular function, except that it *yields* or produces a value.
- A generator `g` yields a value whenever we call `next(g)` on it. It is then *temporarily suspended* until we call `next(g)` again (it is *lazy* and executes *on demand*).
- A `StopIteration` is raised when we reach a `return` statement or the end of the generator function body.

GENERATOR FUNCTION

Any `python` function that has the `yield` keyword in its body is a *generator function*. A generator function is a function which, when called, returns a *generator object*. You can think of generator functions as *factories* for generator objects.

Example:

```
1 def gen_107():
2     """Generator function"""
3     yield 1
4     yield 0
5     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.

```
 1 >>> from inspect import getgeneratorstate
 2 >>> gen_107
 3 <function gen_107 at 0x7fe79f4e7310>
 4 >>> g = gen_107()
 5 <generator object gen_107 at 0x7fe79f57d820>
 6 >>> getgeneratorstate(g)
 7 'GEN_CREATED'
 8 >>> next(g)
 9 1
10 >>> getgeneratorstate(g)
11 'GEN_SUSPENDED'
12 >>> list(g)
13 [0, 7]
14 >>> getgeneratorstate(g)
15 'GEN_CLOSED'
```

GENERATOR FUNCTION

Example:

```
1 def gen_107():
2     """Generator function"""
3     yield 1
4     yield 0
5     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.

```
1 >>> for i in gen_107():
2 ...     i
3 ...
4 1
5 0
6 7
```

A generator implements the standard iterator interface. We can use it *interchangeably* with iterators!

A generator function creates separate instances of generators (it is a factory):

```
1 >>> a, b = gen_107(), gen_107()
2 >>> a; b
3 <generator object gen_107 at 0x7fe79f2cf120>
4 <generator object gen_107 at 0x7fe79f2cf190>
5 >>> iter(a); iter(b)
6 <generator object gen_107 at 0x7fe79f2cf120>
7 <generator object gen_107 at 0x7fe79f2cf190>
```

GENERATOR FUNCTION

Example:

```
1 def gen_107():
2     """Generator function"""
3     yield 1
4     yield 0
5     return
6     yield 7
```

```
1 >>> for i in gen_107():
2 ...     i
3 ...
4 1
5 0
```

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

Generator states: obtained by `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

REVISIT ITERATOR FOR LINKED LIST

Iterator implementation for linked list in previous class:

```
 1 class LinkedList:
 2     # other code skipped
 3
 4     def __iter__(self):
 5         return LinkedListForwardIterator(
 6             self.first)
 7
 8 class LinkedListForwardIterator:
 9     def __init__(self, start):
10         self.node = start
11
12     def __next__(self):
13         if self.node != None:
14             curr_node = self.node
15             self.node = self.node.next
16             return curr_node
17         else:
18             raise StopIteration
19
20     def __iter__(self):
21         return self
```

Same functionality implemented with a generator function:

```
1 class LinkedList:
2     # other code skipped
3
4     def __iter__(self):
5         node = self.first
6         while node != None:
7             yield node
8             node = node.next
```

- `__iter__` is now a generator function (a factory of generators).
- A generator implements the standard iterator interface.
- No need to write an extra class for it! Less code, 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 for x in range(0))
<generator object <genexpr> at 0x7f6c2d732430>
```

- You can not 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)
```

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 (eager):

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

Output (lazy):

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

GENERATOR EXPRESSIONS

- Generator expressions are syntactic sugar. They are nice in places where you want to be brief and concise, same as with comprehensions.
- Generator functions and generator expressions both return generators. They are both generator factories and perform the same job.
- Generator functions are much more flexible and allow for multiple statements and more complex code. Generator functions can further be used as *coroutines* (will be introduced later).
- Generator expressions were proposed in PEP 289.

GENERATOR FUNCTIONS IN THE STANDARD LIBRARY

- The `python` standard library has many generator utilities implemented.
- You should be aware of them in order not to reinvent the wheel.
- Categories include:
  - Filters
  - Maps
  - Merge of inputs
  - Expansion of input into multiple outputs
  - Rearrangements
  - Reductions
- Most of these tools are available in the `itertools` module the others are built-in.

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 a very powerful tool
- Makes it trivial to recursively iterate over a file system tree.

GENERATOR FUNCTIONS IN THE STANDARD LIBRARY

Example: `filter`

- Filter elements in an iterable for which a *predicate function* returns true.
- `filter` returns an iterator.
- Examples: filter vowels

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

  or filter types:

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

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

```
1 >>> from itertools import filterfalse
2 >>> list(filterfalse(lambda x: isinstance(x, int), [0, 0x1, 0o2, 3.0]))
3 [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.
- Example:

```
1 >>> list(map(lambda a, b: (a, b), range(11), list('ABC')))
2 [(0, 'A'), (1, 'B'), (2, 'C')]
```

  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

- Another case where generators are useful is to define different readers of input data which must expose an identical interface to retrieve the data (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, that is, lazily read the data from the disk or tape. A generator is the right tool for this.
- Assume you are working with the following function to process data, where your input data may either come from an ASCII text file or a NumPy binary file (due to different data collection procedures):

```
 1 from itertools import chain
 2 def process(*data_generators):
 3     val = 0
 4     item_count = 0
 5     for item in chain.from_iterable(data_generators):
 6         val += item
 7         item_count += 1
 8         if item_count % 10000 == 0:
 9             print(f'{item_count} items processed from input')
10     return val
```

APPLICATION EXAMPLE: LAZY READ OF LARGE DATA

- Data processing function:

```
 1 from itertools import chain
 2 def process(*data_generators):
 3     val = 0
 4     item_count = 0
 5     for item in chain.from_iterable(data_generators):
 6         val += item
 7         item_count += 1
 8         if item_count % 10000 == 0:
 9             print(f'{item_count} items processed from input')
10     return val
```

- The `process()` function takes a number of data generators (or just iterables) and chains them together to iterate over individual data items that must be processed. Examples could be movie frames, pressure fields from simulations, audio samples, sentences, anything really. In the example above the generators yield just one number at a time.
- If the data file(s) are larger than your physical RAM size or you read from a continuous stream, you will not be able to load the full data set and you must use a *generator* instead.

APPLICATION EXAMPLE: LAZY READ OF LARGE DATA

Example lazy binary reader for some data format irrelevant for this example:

```
def lazy_binary(fname):
    """Lazy binary load (generator)"""
    with open(fname, 'r') as fin:
        while True:
            x = np.fromfile(
                fin, dtype=float, count=1)
            if x.size > 0:
                yield x
            else:
                break
```

Example eager binary reader version. This loads all the data into RAM:

```
def eager_binary(fname):
    """Eager binary load (iterable numpy array)"
    with open(fname, 'r') as fin:
        return np.fromfile(fin, dtype=float)
```

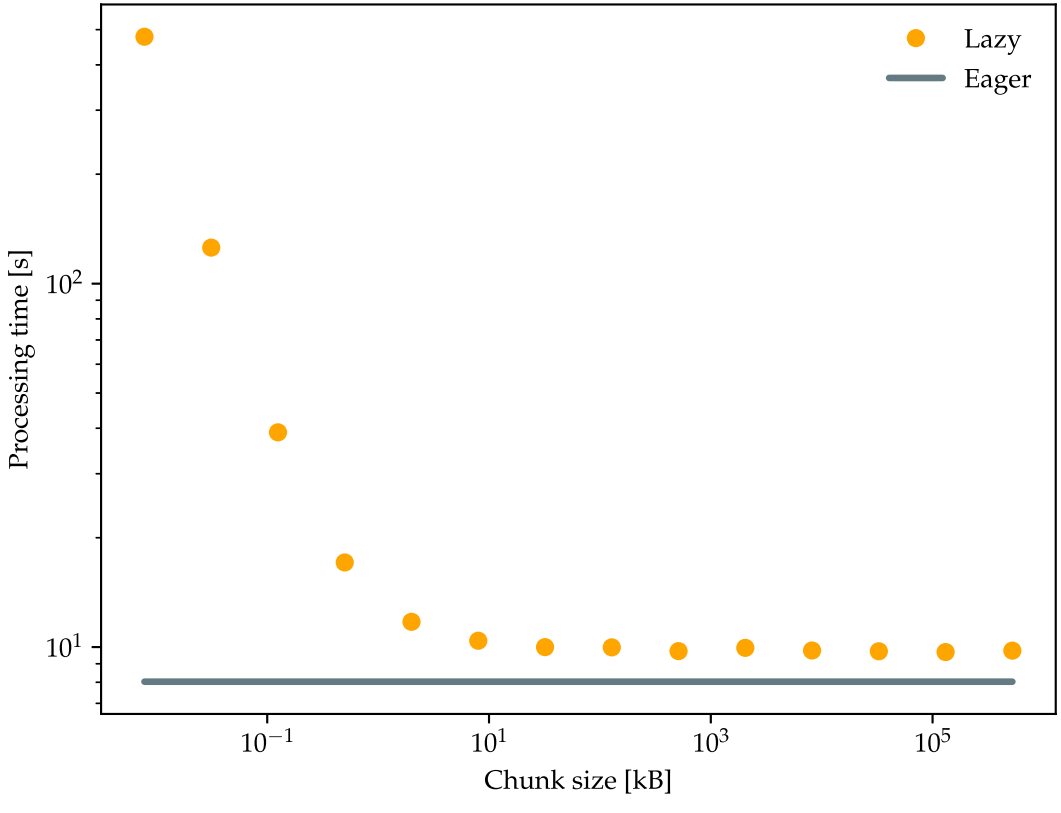
- Be aware that loading small chunks of data from disk is significantly slower than reading it from RAM.

If you must read from disk (lazy) and your individual data elements are small, consider reading larger chunks at a time.

```
def lazy_binary(fname, chunk_size=1):
    """Lazy binary load (generator)"""
    with open(fname, 'r') as fin:
        while True:
            x = np.fromfile(
                fin, dtype=float,
                count=chunk_size)
            for i in x:
                yield i
            if x.size == 0:
                break
```

APPLICATION EXAMPLE: LAZY READ OF LARGE DATA

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



- If you lazy read data that is very small, you can end up spending 100x longer processing your data.
- You should ***at least*** read 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 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 *coroutines*. Functions are *asymmetric* between caller and callee, coroutines are *symmetric*. 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 the `yield` keyword in its body.
- In a coroutine the `yield` keyword usually appears on the right side of an expression, for example

```
def coroutine():
    while True:
        x = yield
        print(x)
```

and it *may or may not* produce a value. If there is no expression after the `yield`, like above, it yields `None`.

- Unlike a generator, a coroutine *can receive data from the caller* by calling `c.send(data)` instead of `next(c)`:

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

COROUTINES EXAMPLE

```
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
14     print(getgeneratorstate(c))
15
16     c.send('Hello')
17     print('main(): control back in main function')
18     last_call = c.send('CS107')
19     print(f'main(): called coroutine() {last_call} times'
20
21     c.close()
22     print(getgeneratorstate(c))
23
24 if __name__ == "__main__":
25     main()
```

```
1 GEN_CREATED
2 GEN_SUSPENDED
3 coroutine(): Hello (call id: 1)
4 main(): control back in main function
5 coroutine(): CS107 (call id: 2)
6 main(): called coroutine() 2 times
7 GEN_CLOSED
```

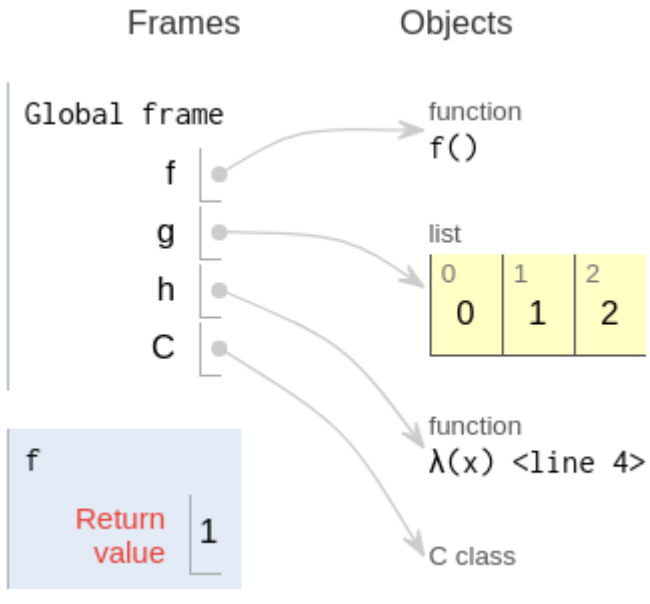
- `line 11` creates the coroutine (noting has been run yet).
- `line 13` primes the coroutine. This means activate it and run until the first `yield`, then suspend it.
- `line 16` sends data to the suspended coroutine. This will activate it and run until the next `yield` is reached. The `.send()` call 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 that 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:*** the linked list `__iter__` we discussed, see PEP 255).
  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 (mental picture: a *stack* data structure, the blue shaded frame is at the top of the stack).
- ***Objects:*** any other `python` objects. Functions, classes, data, etc.

- There are a stacked *sequence* of frames that *execute* code.
- Arrows indicate *references* to objects *in memory*.
- When we enter a function `f()`, a new *frame* appears that executes the code of that function.
- When done the function frame disappears and we enter the caller frame again (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 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* are *references* to that a specific location.
- The *type* of an object describes the internal representation as well as methods and operations it supports.
- 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`.

```
1 >>> def f():
2 ...     pass
3 ...
4 >>> g = lambda x: x
5 >>> f.__code__; g.__code__
6 <code object f at 0x7fd88a3fac90, file "<stdin>", line 1>
7 <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 you write gets ***compiled*** into *bytecode* objects *on the fly*.
- `python` is an *interpreted* language, under the hood, 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