

The Fibonacci sequence

Rachid Hamadi, CSE, UNSW

COMP9021 Principles of Programming, Term 3, 2019

```
[1]: from functools import lru_cache
```

The Fibonacci sequence, say $(F_n)_{n \in \mathbb{N}}$, is defined as $F_0 = 0, F_1 = 1$ and for all $n > 1, F_n = F_{n-2} + F_{n-1}$; so it is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34...

A generator function is the best option to generate the initial segment of the Fibonacci sequence of a given length, even though it can also be used to generate the member of the Fibonacci sequence of a given index:

```
[2]: def fibonacci_sequence():
    yield 0
    yield 1
    previous, current = 0, 1
    while True:
        previous, current = current, previous + current
        yield current
```

```
[3]: S = fibonacci_sequence()
list(next(S) for _ in range(19))
```

```
[3]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584]
```

```
[4]: from IPython.display import clear_output

S = fibonacci_sequence()
for _ in range(18):
    next(S)
    clear output()
next(S)
```

```
[4]: 2584
```

In case only one or a few specific members of the Fibonacci sequence are needed, a simple function is more appropriate:

```
[5]: def iterative_fibonacci(n):
    if n < 2:
        return n
    previous, current = 0, 1
    for _ in range(2, n + 1):
        previous, current = current, previous + current
    return current
```

```
iterative_fibonacci(18)
```

[5]: 2584

A naive recursive implementation is elegant, but too inefficient, as we will see:

```
[6]: def recursive_fibonacci(n):  
    if n >= 2:  
        return recursive_fibonacci(n - 2) + recursive_fibonacci(n - 1)  
    return n  
  
recursive_fibonacci(18)
```

[6]: 2584

Let an integer n greater than 1 be given. Then a call to `recursive_fibonacci(n)` involves:

- for all nonzero $k \leq n$, F_{n-k+1} calls to `recursive_fibonacci(k)`;
- F_{n-1} calls to `recursive_fibonacci(0)`.

In particular, `recursive_fibonacci(n)` calls `recursive_fibonacci(1)` F_n times. Proof is by induction on $k \leq n$:

- `recursive_fibonacci(n)` is called once indeed.
- `recursive_fibonacci(n)` directly calls `recursive_fibonacci(n - 1)` and does not call it indirectly, so calls it once indeed.
- For all $k < n$, `recursive_fibonacci(n - k)` is directly called by `recursive_fibonacci(n - k + 1)` and by `recursive_fibonacci(n - k + 2)`. By inductive hypothesis, the latter two are called directly or indirectly by `recursive_fibonacci(n)` F_k and F_{k-1} times, respectively. Hence `recursive_fibonacci(n - k)` is called by `recursive_fibonacci(n)` F_{k+1} times.
- `recursive_fibonacci(0)` is directly called by `recursive_fibonacci(2)`, hence it is called by `recursive_fibonacci(n)` F_{n-1} times.

Let us illustrate this for $n = 6$ with the following tracing function:

```
[7]: def trace_recursive_fibonacci(n, depth):  
    print('    ' * depth, 'Start of function call for n =', n)  
    if n >= 2:  
        second_previous = trace_recursive_fibonacci(n - 2, depth + 1)  
        previous = trace_recursive_fibonacci(n - 1, depth + 1)  
        print('    ' * depth, f'End of function call for n = {n}, returning',  
              second_previous + previous  
              )  
        return second_previous + previous  
    print('    ' * depth, f'End of function call for n = {n}, returning', n)  
    return n  
  
trace_recursive_fibonacci(6, 0)
```

Start of function call for n = 6

Start of function call for n = 4

```

Start of function call for n = 2
  Start of function call for n = 0
  End of function call for n = 0, returning 0
  Start of function call for n = 1
  End of function call for n = 1, returning 1
End of function call for n = 2, returning 1
Start of function call for n = 3
  Start of function call for n = 1
  End of function call for n = 1, returning 1
  Start of function call for n = 2
  Start of function call for n = 0
  End of function call for n = 0, returning 0
  Start of function call for n = 1
  End of function call for n = 1, returning 1
  End of function call for n = 2, returning 1
End of function call for n = 3, returning 2
End of function call for n = 4, returning 3
Start of function call for n = 5
  Start of function call for n = 3
  Start of function call for n = 1
  End of function call for n = 1, returning 1
  Start of function call for n = 2
  Start of function call for n = 0
  End of function call for n = 0, returning 0
  Start of function call for n = 1
  End of function call for n = 1, returning 1
  End of function call for n = 2, returning 1
End of function call for n = 3, returning 2
Start of function call for n = 4
  Start of function call for n = 2
  Start of function call for n = 0
  End of function call for n = 0, returning 0
  Start of function call for n = 1
  End of function call for n = 1, returning 1
  End of function call for n = 2, returning 1
Start of function call for n = 3
  Start of function call for n = 1
  End of function call for n = 1, returning 1
  Start of function call for n = 2
  Start of function call for n = 0
  End of function call for n = 0, returning 0
  Start of function call for n = 1
  End of function call for n = 1, returning 1
  End of function call for n = 2, returning 1
  End of function call for n = 3, returning 2
End of function call for n = 4, returning 3
End of function call for n = 5, returning 5
End of function call for n = 6, returning 8

```

[7]: 8

We can still save the recursive design by saving terms of the Fibonacci sequence as they get computed for the first time. As a result of processing the `def` statement below, a dictionary, `fibonacci`, is created and initialised with the values of the first two members of the Fibonacci sequence. Then the function `memoise_fibonacci()` is called, directly as `memoise_fibonacci(18)`, and indirectly as `memoise_fibonacci(18)` executes. For each of those calls, `memoise_fibonacci()` is given one argument only, so the second argument is set to its default value, namely, `fibonacci`, extended with a new key and associated value in case the condition of the `if` statement in the body of `memoise_fibonacci()` evaluates to `True`:

```
[8]: def memoise_fibonacci(n, fibonacci={0: 0, 1: 1}):
      if n not in fibonacci:
          fibonacci[n] = memoise_fibonacci(n - 2) + memoise_fibonacci(n - 1)
      return fibonacci[n]

memoise_fibonacci(18)
```

[8]: 2584

Let us illustrate the mechanism for $n = 6$ with the following tracing function:

```
[9]: def trace_memoise_fibonacci(n, depth, fibonacci={0: 0, 1: 1}):
      print('    ' * depth, 'Start of function call for n =', n)
      print('    ' * (depth + 1), f'fibonacci now is {fibonacci}; ', end = '')
      if n not in fibonacci:
          print('compute value')
          fibonacci[n] = trace_memoise_fibonacci(n - 2, depth + 1)\
                          + trace_memoise_fibonacci(n - 1, depth + 1)
      else:
          print('retrieve value')
      print('    ' * depth, f'End of function call for n = {n}, returning',
            fibonacci[n])
      return fibonacci[n]

trace_memoise_fibonacci(6, 0)
```

```
Start of function call for n = 6
  fibonacci now is {0: 0, 1: 1}; compute value
  Start of function call for n = 4
    fibonacci now is {0: 0, 1: 1}; compute value
    Start of function call for n = 2
      fibonacci now is {0: 0, 1: 1}; compute value
      Start of function call for n = 0
        fibonacci now is {0: 0, 1: 1}; retrieve value
        End of function call for n = 0, returning 0
      Start of function call for n = 1
        fibonacci now is {0: 0, 1: 1}; retrieve value
        End of function call for n = 1, returning 1
```

```

End of function call for n = 2, returning 1
Start of function call for n = 3
    fibonacci now is {0: 0, 1: 1, 2: 1}; compute value
    Start of function call for n = 1
        fibonacci now is {0: 0, 1: 1, 2: 1}; retrieve value
    End of function call for n = 1, returning 1
    Start of function call for n = 2
        fibonacci now is {0: 0, 1: 1, 2: 1}; retrieve value
    End of function call for n = 2, returning 1
End of function call for n = 3, returning 2
End of function call for n = 4, returning 3
Start of function call for n = 5
    fibonacci now is {0: 0, 1: 1, 2: 1, 3: 2, 4: 3}; compute value
    Start of function call for n = 3
        fibonacci now is {0: 0, 1: 1, 2: 1, 3: 2, 4: 3}; retrieve value
    End of function call for n = 3, returning 2
    Start of function call for n = 4
        fibonacci now is {0: 0, 1: 1, 2: 1, 3: 2, 4: 3}; retrieve value
    End of function call for n = 4, returning 3
End of function call for n = 5, returning 5
End of function call for n = 6, returning 8

```

[9]: 8

`memoise_fibonacci()` illustrates the fact that when a function argument has a default value, that default value is not created at every function call, but at the time when Python processes the function's `def` statement. This makes no difference for default values of a type such as `int`:

```

[10]: def f(x=0):
        x += 1
        return x

# Create the argument 0 before calling f(), let x denote it, from the
# value denoted by x and 1 create 1, let x denote it.
f(0)
f(1)
f(2)
# Let x denote the 0 created when def was processed, from the value
# denoted by x and 1 create 1, let x denote it.
f()
f()
f()

```

[10]: 1

[10]: 2

[10]: 3

[10]: 1

[10]: 1

[10]: 1

But it makes a difference for default values of a type such as list:

```
[11]: def g(x=[0]):  
      x += [1]  
      return x  
  
      # Create the argument [0] before calling g(), let x denote it, then  
      # extend it to [0, 1], let x denote the modified list.  
g([0])  
g([1])  
g([2])  
# Let x denote the list L created when def was processed, then and now  
# equal to [0], then extend it to [0, 1], let x denote the modified L.  
g()  
# Let x denote the list L created when def was processed, now equal to  
# [0, 1], then extend it to [0, 1, 1], let x denote the modified L.  
g()  
g()
```

[11]: [0, 1]

[11]: [1, 1]

[11]: [2, 1]

[11]: [0, 1]

[11]: [0, 1, 1]

[11]: [0, 1, 1, 1]

What was good for `memoise_fibonacci()` might not be the intended behaviour for other functions, in other contexts: in case a function F is called without an argument for a parameter p that in F 's definition, receives a default value v , one might want p to always be assigned that default value, not the value currently denoted by p and possibly modified from the original value of v following previous calls to F . One should then opt for a different design:

```
[12]: def h(x=None):  
      if x is None:  
          x = [0]  
      x += [1]  
      return x  
  
      # Create the argument [0] before calling h(), let x denote it, then  
      # extend it to [0, 1], let x denote the modified list.  
h([0])  
h([1])  
h([2])  
# Let x denote None, then create [0], let x denote it, then extend it to
```

```
# [0, 1], let x denote the modified list.  
h()  
h()  
h()
```

[12]: [0, 1]

[12]: [1, 1]

[12]: [2, 1]

[12]: [0, 1]

[12]: [0, 1]

[12]: [0, 1]

The `lru_cache()` ("lru" is for *Least Recently Used*) function from the `functools` module returns a function that can be used as a **decorator** and applied to a function F to yield a memoised version of F . By default, the `maxsize` argument of `lru_cache()` is set to 128, to record up to the last 128 computed values of the function, as witnessed by the `cache_info()` attribute of the memoised version of f :

```
[13]: @lru_cache()  
def lru_fibonacci(n):  
    if n < 2:  
        return n  
    return lru_fibonacci(n - 1) + lru_fibonacci(n - 2)  
  
lru_fibonacci.cache_info()
```

[13]: CacheInfo(hits=0, misses=0, maxsize=128, currsize=0)

Suppose that `lru_fibonacci()` is called for the first time with 2 as argument. Since `lru_fibonacci(2)` has not been computed yet, `lru_fibonacci(1)` and `lru_fibonacci(0)` are called, which both have not been computed yet either: a total of 3 values fail to be retrieved (3 misses). The last two values are computed and recorded, then the former value is computed and recorded, and the cache eventually stores those 3 values:

```
[14]: lru_fibonacci(2)  
lru_fibonacci.cache_info()
```

[14]: 1

[14]: CacheInfo(hits=0, misses=3, maxsize=128, currsize=3)

Calling `lru_fibonacci(2)` again, the value is found in the cache (1 hit):

```
[15]: lru_fibonacci(2)  
lru_fibonacci.cache_info()
```

[15]: 1

[15]: CacheInfo(hits=1, misses=3, maxsize=128, currsize=3)

When calling `lru_fibonacci(3)`, the value fails to be found in the cache (1 more miss), so `lru_fibonacci(2)` and `lru_fibonacci(1)` are called and retrieved from the cache (2 more hits), and the computed value of `lru_fibonacci(3)` is added to the cache:

```
[16]: lru_fibonacci(3)
lru_fibonacci.cache_info()
```

```
[16]: 2
```

```
[16]: CacheInfo(hits=3, misses=4, maxsize=128, currsz=4)
```

The cache can be cleared with the `cache_clear()` attribute of the memoised version of the function. Then calling `lru_fibonacci(3)` necessitates to call `lru_fibonacci(2)` and `lru_fibonacci(1)`, calling `lru_fibonacci(2)` necessitates to call `lru_fibonacci(1)` and `lru_fibonacci(0)`, for a total of 4 misses that are computed and all stored in the cache:

```
[17]: lru_fibonacci.cache_clear()
lru_fibonacci(3)
lru_fibonacci.cache_info()
```

```
[17]: 2
```

```
[17]: CacheInfo(hits=1, misses=4, maxsize=128, currsz=4)
```

Clearing the cache again, calling `lru_fibonacci(128)` necessitates to call for the first time `lru_fibonacci(128)`, ..., `lru_fibonacci(0)` (129 misses). When calling `lru_fibonacci(2)` for the first time, `lru_fibonacci(1)` could be called before `lru_fibonacci(0)` or the other way around. Execution of the following cell reveals that `lru_fibonacci(0)` is called first; its value leaves the cache after the values of `lru_fibonacci(1)`, ..., `lru_fibonacci(128)` have then been computed and recorded. When `lru_fibonacci(3)` is computed, `lru_fibonacci(1)` is retrieved (whether `lru_fibonacci(1)` or `lru_fibonacci(2)` is computed first), ..., when `lru_fibonacci(128)` is computed, `lru_fibonacci(126)` is retrieved (whether `lru_fibonacci(126)` or `lru_fibonacci(127)` is computed first), for a total of 126 hits:

```
[18]: lru_fibonacci.cache_clear()
lru_fibonacci(128)
lru_fibonacci.cache_info()
lru_fibonacci(1)
lru_fibonacci.cache_info()
lru_fibonacci(0)
lru_fibonacci.cache_info()
```

```
[18]: 251728825683549488150424261
```

```
[18]: CacheInfo(hits=126, misses=129, maxsize=128, currsz=128)
```

```
[18]: 1
```

```
[18]: CacheInfo(hits=127, misses=129, maxsize=128, currsz=128)
```

```
[18]: 0
```

```
[18]: CacheInfo(hits=127, misses=130, maxsize=128, currsz=128)
```

The capacity of the cache can be left unbounded by setting the value of the `maxsize` argument of `lru_cache()` to `None`:

```
[19]: @lru_cache(None)
def unbounded_lru_fibonacci(n):
    if n < 2:
```



```
    return n
    return unbounded_lru_fibonacci(n - 1) + unbounded_lru_fibonacci(n - 2)
```

```
[20]: unbounded_lru_fibonacci(150)
      unbounded_lru_fibonacci.cache_info()
```

```
[20]: 9969216677189303386214405760200
```

```
[20]: CacheInfo(hits=148, misses=151, maxsize=None, currsize=151)
```

The argument `maxsize` of `lru_cache()` can also be set to any integer value. Let us set it to 4 and first call `bounded_lru_fibonacci(8)`. Then `bounded_lru_fibonacci(8)`, `bounded_lru_fibonacci(7)`, `bounded_lru_fibonacci(6)` and `bounded_lru_fibonacci(5)` are last called and recorded. If `bounded_lru_fibonacci(5)` is then called, its value is retrieved (1 more hit). And if `bounded_lru_fibonacci(4)` is thereafter called, `bounded_lru_fibonacci(4)`, ..., `bounded_lru_fibonacci(0)` have to be recomputed (5 more misses), with `bounded_lru_fibonacci(2)` and `bounded_lru_fibonacci(1)` being retrieved in the process (2 more hits):

```
[21]: @lru_cache(4)
      def bounded_lru_fibonacci(n):
          if n < 2:
              return n
          return bounded_lru_fibonacci(n - 1) + bounded_lru_fibonacci(n - 2)
```

```
[22]: bounded_lru_fibonacci(8)
      bounded_lru_fibonacci.cache_info()
      bounded_lru_fibonacci(5)
      bounded_lru_fibonacci.cache_info()
      bounded_lru_fibonacci(4)
      bounded_lru_fibonacci.cache_info()
```

```
[23]: 21
```

```
[23]: CacheInfo(hits=6, misses=9, maxsize=4, currsize=4)
```

```
[23]: 5
```

```
[23]: CacheInfo(hits=7, misses=9, maxsize=4, currsize=4)
```

```
[23]: 3
```

```
[23]: CacheInfo(hits=9, misses=14, maxsize=4, currsize=4)
```