The University of Texas at San Antonio™

EE-3233, Lab Lecture 3

Functions and Recursion in Python3

Functions

- In a similar fashion to C, or any other programming language, you can define functions in Python3.
- You define them via the **`def`** operator, and with the following syntax:

```
def <name-of-function>(<param-1>, <param-2>, …, <param-n>):
```

Parameters

- Parameters are passed by value, meaning that a copy is made for the function, unless it is the instance of an object.
- For example, consider that x is an int:

```
def add_one(n):
    n += 1 # Adds one to n in the stack.

x = 0
add_one(x)
print(x) # Prints 0.
```

Parameters

- On the contrary, if an object is passed, then the object will be passed as a reference (like passing a pointer in C):

```
def append_one(l):
   l.append(1) # Append an integer (1) to the end of list l.
x = [] # This is an empty list.
append_one(x)
print(x) # Prints [1].
```

Parameters

- To make code more readable, you can specify the data type for each parameter passed to a function:

```
def <name-of-function>(<param>: <data-type>):
```

- However, note that Python3 won’t enforce that <param> is the of the specified <data-type>, this is more like a note.

Parameters: Extra

- Pass multiple keyword args to a function without predetermining the total number of args

```
def unlimited_args(*tuple_args):
    …
    return
```

- Pass multiple keyword args to a function in a dictionary

```
def unlimited_args(**dict_args):
    …
    return
```

[How to pass multiple arguments to function ? - GeeksforGeeks]()

Return Value

- If by the end of a function there is not a **`return`** statement with a value, the function by default returns **`None`** (nothing).

- Also, for readability, it is possible to specify a return data type:

```
def add(a: int, b: int) -> int:
    return a + b
```

Recursion

- Recursion is a technique or strategy to solve a complex problems by combining the solutions of subproblems.
- The key here is that a recursive function repeatedly calls itself until it reaches a *base case* where the solution to the problem is trivial. At this point, the function returns, and the previous calls of the recursive function use the partial solution to create a complete solution.
- A perfect example to demonstrate the principle behind recursion is the Fibonacci sequence.

Example: The Fibonacci Sequence

Index (n)	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Fibonacci No. (F(n))	0	1	1	2	3	5	8

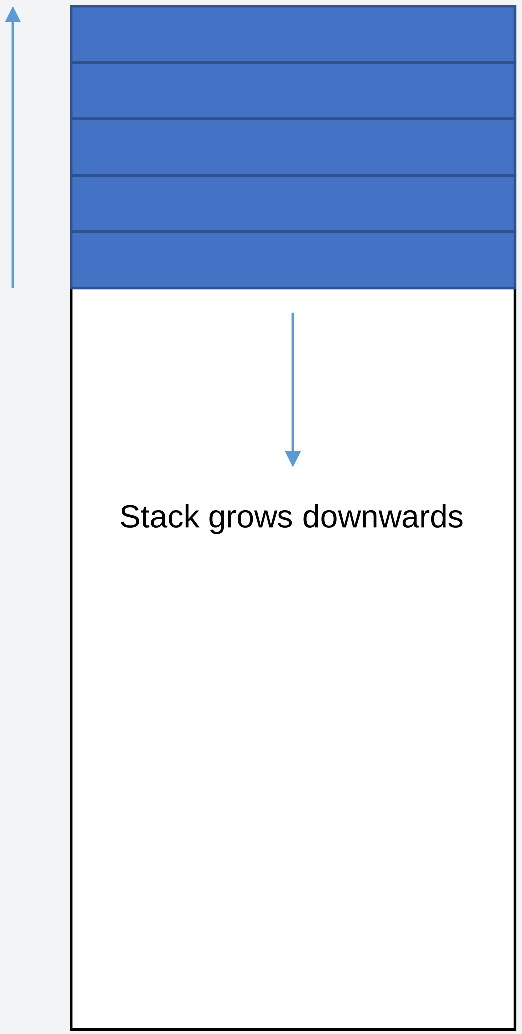
…
This continues to infinity

Relationship:

```
if n < 2:
        F(n) = n # Base case.
else:
        F(n) = F(n – 1) + F(n – 2) # Recursive calls to F(n).
```

Virtual Address Space

By combining the solutions of
F(n -1) + F(n -2),
the stack starts decrease.



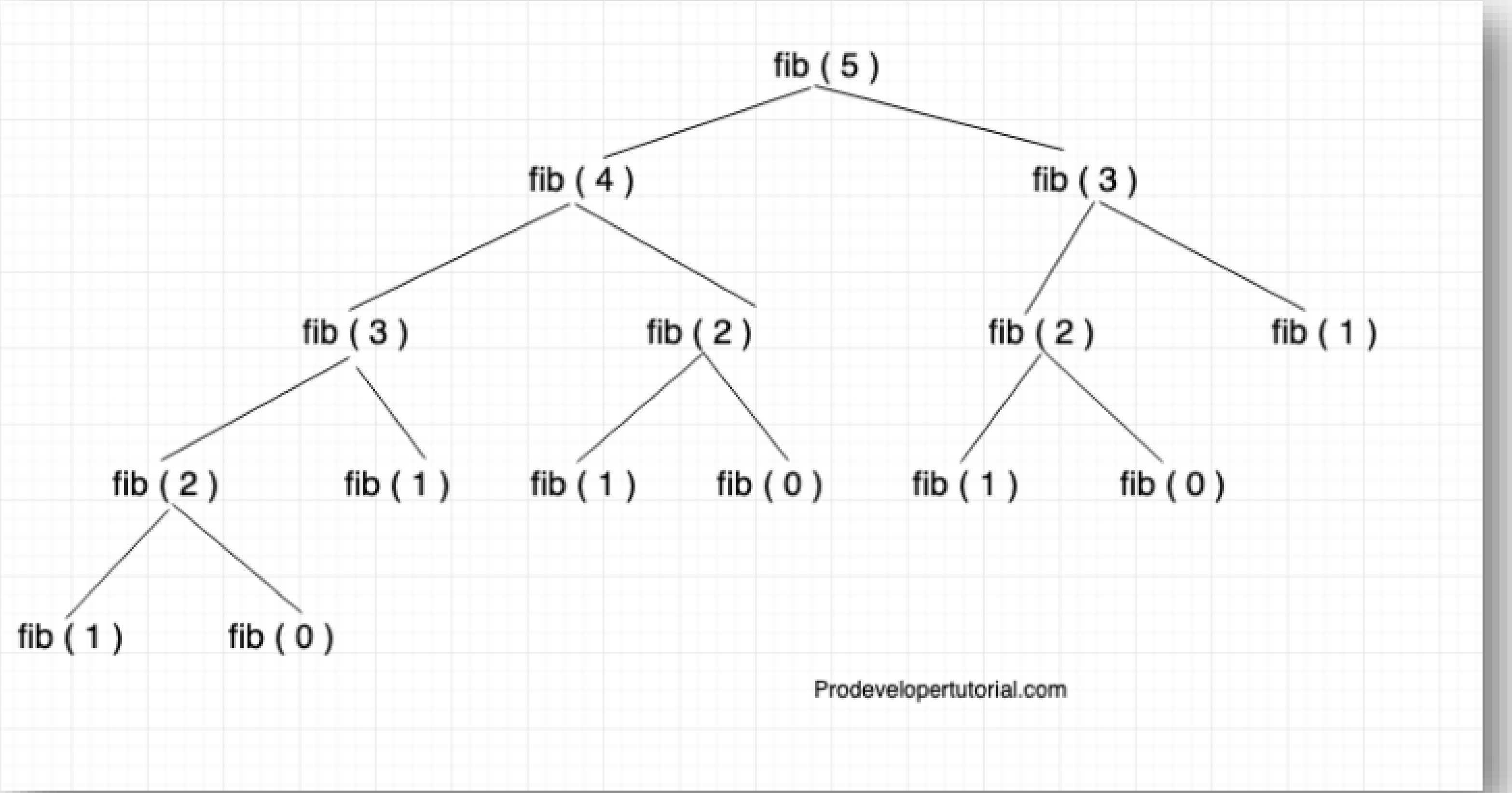
The height of the stack is
directly proportional to the value of n.

When a base case is reached
the F(n) call on top of the stack
can return immediately.

Fibonacci implementation (Naïve)

```
def fibonacci(n: int) -> int:
    if n < 2:
        return n
    else:
        return fibonacci(n – 1) + fibonacci(n – 2)
```

Fibonacci implementation: Recursion Tree



Optimizing Fibonacci via Memoization

- However, there is a problem. If coded naively and with big values for n, the Fibonacci function (F(n)) will reach the base case around n^2 times! This occurs because F(n) constantly computes the result of values (n – m) multiple times. In other words, F(n) keeps doing redundant work.

- To solve this, it is possible to save values F(n) that have already been computed in a hashmap. This technique is called memorization.

Dictionaries

- Python implementation of hashmaps (unordered)
- Access time is constant O(1) compared to ‘list’ data structure O(n)
- Elements are composed of key, values pair
- Syntax:

```
newdict = {} or newdict = dict()       # creates empty dict
newdict = {'key': 'some value', 1: 100, 'list_key': [1,2,3] } # creates and initializes
newdict.keys()         # get all keys available in dict, returns list
newdict.values()       # get all values available for all keys, returns list
newdict[‘new key’] = ‘Adding a new key,value pair to dict’
some_variable = newdict[1]            # retrieve contents associated with key 1
newdict[‘key not in dict’]            # throws exception ‘KeyError’
```

Nested functions

- Provide encapsulation and hide data from external access
- Same syntax to declare a regular function
- Useful to declare helper inner functions

```
>>> def increment(number): # regular function
...     def inner_increment(): # nested function
...         return number + 1
...     return inner_increment()
...
>>> increment(10)
11
>>> inner_increment(10)
    NameError: name 'inner_increment' is not defined
```

For more info, refer to https://www.codespeedy.com/define-a-function-inside-a-function-in-python/

utsa.edu