

Lecture 11, 19 September 2024

- Recursive definition of factorial
- Add diagnostic `print()` to trace recursive calls

```
In [1]: def fact(n):  
        print("evaluating factorial of",n)  
        if n == 1:  
            return(1)  
        else:  
            return(n*fact(n-1))
```

```
In [2]: fact(8)
```

```
evaluating factorial of 8  
evaluating factorial of 7  
evaluating factorial of 6  
evaluating factorial of 5  
evaluating factorial of 4  
evaluating factorial of 3  
evaluating factorial of 2  
evaluating factorial of 1
```

```
Out[2]: 40320
```

- A negative argument creates an unending sequence of recursive calls
- Python has a built in `recursion limit` to abort such cases
 - This limit may deny some legitimate computation -- for instance `factorial(3000)` will also abort
 - We will see later that we can increase this limit

```
In [3]: fact(-1)
```



```
evaluating factorial of -2925
evaluating factorial of -2926
evaluating factorial of -2927
evaluating factorial of -2928
evaluating factorial of -2929
evaluating factorial of -2930
evaluating factorial of -2931
evaluating factorial of -2932
evaluating factorial of -2933
evaluating factorial of -2934
evaluating factorial of -2935
evaluating factorial of -2936
evaluating factorial of -2937
evaluating factorial of -2938
evaluating factorial of -2939
evaluating factorial of -2940
evaluating factorial of -2941
evaluating factorial of -2942
evaluating factorial of -2943
evaluating factorial of -2944
evaluating factorial of -2945
evaluating factorial of -2946
evaluating factorial of -2947
evaluating factorial of -2948
evaluating factorial of -2949
evaluating factorial of -2950
evaluating factorial of -2951
evaluating factorial of -2952
evaluating factorial of -2953
evaluating factorial of -2954
evaluating factorial of -2955
evaluating factorial of -2956
evaluating factorial of -2957
evaluating factorial of -2958
evaluating factorial of -2959
evaluating factorial of -2960
evaluating factorial of -2961
evaluating factorial of -2962
evaluating factorial of -2963
evaluating factorial of -2964
evaluating factorial of -2965
evaluating factorial of -2966
evaluating factorial of -2967
evaluating factorial of -2968
evaluating factorial of -2969
```

```
-----
RecursionError                                Traceback (most recent call last)
Cell In[3], line 1
----> 1 fact(-1)

Cell In[1], line 6, in fact(n)
      4     return(1)
      5 else:
----> 6     return(n*fact(n-1))

Cell In[1], line 6, in fact(n)
      4     return(1)
      5 else:
----> 6     return(n*fact(n-1))

[... skipping similar frames: fact at line 6 (2966 times)]

Cell In[1], line 6, in fact(n)
      4     return(1)
      5 else:
----> 6     return(n*fact(n-1))

Cell In[1], line 2, in fact(n)
      1 def fact(n):
----> 2     print("evaluating factorial of",n)
      3     if n == 1:
      4         return(1)

File ~/python-venv/venv/lib/python3.11/site-packages/ipykernel/iostream.py:681, in OutStream.write(self, string)
    678     msg = "I/O operation on closed file"
    679     raise ValueError(msg)
--> 681 is_child = not self._is_master_process()
    682 # only touch the buffer in the IO thread to avoid races
    683 with self._buffer_lock:

File ~/python-venv/venv/lib/python3.11/site-packages/ipykernel/iostream.py:551, in OutStream._is_master_process(self)
    550 def _is_master_process(self):
--> 551     return os.getpid() == self._master_pid

RecursionError: maximum recursion depth exceeded while calling a Python object
```

- We can also induct on the structure of a datastructure
- We can break up a list into the first element (the *head*) and the rest (the *tail*) as a smaller list
- Here is a recursive definition of `len()`

```
In [4]: def mylen(l):
        print("Call with",l)
        if l == []:
            return(0)
        else:
            return(1+mylen(l[1:]))
```

```
In [5]: mylen([7,18,2,-1,3])
```

```
Call with [7, 18, 2, -1, 3]
Call with [18, 2, -1, 3]
Call with [2, -1, 3]
Call with [-1, 3]
Call with [3]
Call with []
```

```
Out[5]: 5
```

- A similar definition for `sum()`

```
In [6]: def mysum(l):
        print("Call with",l)
        if l == []:
            return(0)
        else:
            return(l[0]+mysum(l[1:]))
```

```
In [7]: mysum([7,18,2,-1,3])
```

```
Call with [7, 18, 2, -1, 3]
Call with [18, 2, -1, 3]
Call with [2, -1, 3]
Call with [-1, 3]
Call with [3]
Call with []
```

```
Out[7]: 29
```

- Insert a value `v` into a list `l` sorted in ascending order

```
In [8]: def sortedinsert(l,v):
        if v < l[0]:
            return([v]+l)
        else:
            return(l[:1] + sortedinsert(l[1:],v))
```

```
In [9]: sortedinsert(list(range(1,20,2)),14)
```

```
Out[9]: [1, 3, 5, 7, 9, 11, 13, 14, 15, 17, 19]
```

- The built in recursion limit causes this function to fail on moderate size lists

```
In [10]: sortedinsert(list(range(3000)),3000)
```

```
-----  
RecursionError                                Traceback (most recent call last)  
Cell In[10], line 1  
----> 1 sortedinsert(list(range(3000)),3000)  
  
Cell In[8], line 5, in sortedinsert(l, v)  
      3 return([v]+l)  
      4 else:  
----> 5 return(l[:1] + sortedinsert(l[1:],v))  
  
Cell In[8], line 5, in sortedinsert(l, v)  
      3 return([v]+l)  
      4 else:  
----> 5 return(l[:1] + sortedinsert(l[1:],v))  
  
[... skipping similar frames: sortedinsert at line 5 (2970 times)]  
  
Cell In[8], line 5, in sortedinsert(l, v)  
      3 return([v]+l)  
      4 else:  
----> 5 return(l[:1] + sortedinsert(l[1:],v))  
  
RecursionError: maximum recursion depth exceeded
```