What is Tail Recursion?

Thomas Moawad

Classic Approach

Classic

```
def fact(n):
    if n == 1:
        return 1
    return n * fact(n - 1)
```

Stack Trace

```
fact(5)
    -> fact(4)
        -> fact(3)
            -> fact(2)
                -> fact(1)
```

Actual stack trace

VARIABLES

Locals

n: 1

Globals

WATCH

CALL STACK — Paused on breakpoint

fact	example.py	2:1
fact	example.py	4:1
fact	example.py	4:1
fact	example.py	4:1
fact	example.py	4:1
fact	example.py	4:1
fact	example.py	4:1
fact	example.py	4:1
fact	example.py	4:1
fact	example.py	4:1

Users > tommymoawad > Desktop > example.py > fact

```
1  def fact(n):
2      if n == 1:
3          return 1
4      return n * fact(n - 1)
5
6  fact(10)
```

Tail Recursion Approach

New

```
def fact(acc, n):
    if n == 1:
        return acc
    return fact(acc * n, n - 1)
```

Stack Trace

```
fact(1, 5)
    -> fact(5, 4)
        -> fact(20, 3)
            -> fact(60, 2)
                -> fact(120,1)
                    -> 120
```

Tail Recursion Approach

New

```
def fact(acc, n):
    if n == 1:
        return acc
    return fact(acc * n, n - 1)
```

Stack Trace

~~fact(1, 5)~~
~~-> fact(5, 4)~~
~~-> fact(20, 3)~~
~~-> fact(60, 2)~~
~~-> fact(120,1)~~
-> 120

Tail Recursion Approach

New

```
def fact(acc, n):
    if n == 1:
        return acc
    return fact(acc * n, n - 1)
```

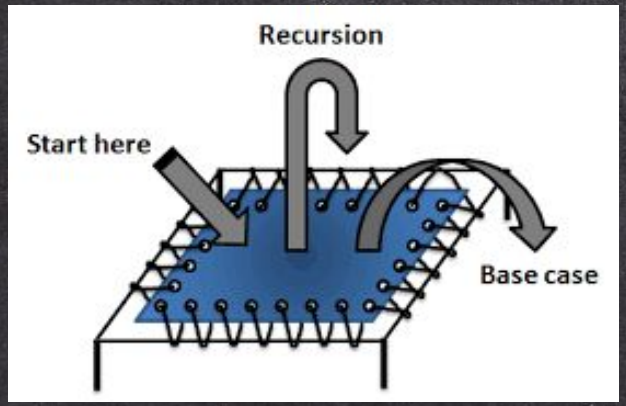
- fact(1, 5) returns fact(5, 4)
- This is like a linked list!
- We only want to keep the tail

Stack Trace

~~fact(1, 5)~~
~~-> fact(5, 4)~~
~~-> fact(20, 3)~~
~~-> fact(60, 2)~~
~~-> fact(120,1)~~
-> 120

Implementation

"Only keep the tail"



Thunk:
The term originated as a whimsical irregular form of the verb *think*. It refers to the original use of thunks in ALGOL 60 compilers, which required special analysis (thought) to determine what type of routine to generate.

```
def trampoline(x):
    while callable(x):
        x = x()
    return x
```

this is a "thunk"

```
def fact(acc, n):
    if n == 1:
        return acc
    return lambda: fact(acc * n, n - 1)
```

VARIABLES

Locals

acc: 1

n: 5

Globals

WATCH

CALL STACK — Paused on breakpoint

fact	example.py	10:1
<module>	example.py	15:1

Users > tommymoawad > Desktop > example.py > fact

```python
# Trampoline: A loop that goes from “thunk” to “thunk”
# Thunk: A function that takes no args and invokes another function
def trampoline(x):
    while callable(x):
        x = x()   # ← this is a “thunk”
    return x


def fact(acc, n):
    print(acc, 1)
    if n == 1:
        return acc
    return lambda: fact(acc * n, n - 1)

print(trampoline(fact(1, 5)))
```

VARIABLES

Locals

acc: 5

n: 4

Globals

WATCH

CALL STACK — Paused on breakpoint

fact	example.py	10:1
<lambda>	example.py	13:1
trampoline	example.py	5:1
<module>	example.py	15:1

Users > tommymoawad > Desktop > example.py > fact

```python
# Trampoline: A loop that goes from “thunk” to “thunk”
# Thunk: A function that takes no args and invokes another function
def trampoline(x):
    while callable(x):
        x = x()   # ← this is a “thunk”
    return x


def fact(acc, n):
    print(acc, 1)
    if n == 1:
        return acc
    return lambda: fact(acc * n, n - 1)

print(trampoline(fact(1, 5)))
```

VARIABLES

Locals

acc: 20

n: 3

Globals

WATCH

CALL STACK

Paused on breakpoint

fact	example.py	10:1
<lambda>	example.py	13:1
trampoline	example.py	5:1
<module>	example.py	15:1
fact	example.py	10:1
<lambda>	example.py	13:1
trampoline	example.py	5:1
<module>	example.py	15:1

Load More Stack Frames

Users > tommymoawad > Desktop > example.py > fact

```python
# Trampoline: A loop that goes from “thunk” to “thunk”
# Thunk: A function that takes no args and invokes another function
def trampoline(x):
    while callable(x):
        x = x()   # ← this is a “thunk”
    return x


def fact(acc, n):
    print(acc, 1)
    if n == 1:
        return acc
    return lambda: fact(acc * n, n - 1)

print(trampoline(fact(1, 5)))
```

VARIABLES

Locals

acc: 60

n: 2

Globals

WATCH

CALL STACK

Paused on breakpoint

fact	example.py	10:1
<lambda>	example.py	13:1
trampoline	example.py	5:1
<module>	example.py	15:1

Users > tommymoawad > Desktop > example.py > fact

```python
# Trampoline: A loop that goes from “thunk” to “thunk”
# Thunk: A function that takes no args and invokes another function
def trampoline(x):
    while callable(x):
        x = x()   # ← this is a “thunk”
    return x


def fact(acc, n):
    print(acc, 1)
    if n == 1:
        return acc
    return lambda: fact(acc * n, n - 1)

print(trampoline(fact(1, 5)))
```

VARIABLES

Locals

acc: 120

n: 1

Globals

WATCH

CALL STACK

Paused on breakpoint

fact	example.py	10:1
<lambda>	example.py	13:1
trampoline	example.py	5:1
<module>	example.py	15:1
fact	example.py	10:1
<lambda>	example.py	13:1
trampoline	example.py	5:1
<module>	example.py	15:1

Load More Stack Frames

Users > tommymoawad > Desktop > example.py > fact

```python
# Trampoline: A loop that goes from “thunk” to “thunk”
# Thunk: A function that takes no args and invokes another function
def trampoline(x):
    while callable(x):
        x = x()   # ← this is a “thunk”
    return x


def fact(acc, n):
    print(acc, 1)
    if n == 1:
        return acc
    return lambda: fact(acc * n, n - 1)

print(trampoline(fact(1, 5)))
```

VARIABLES

Locals

acc: 197606670289682097197512707034994722148841387889567404713793803735946308692246198336411101660…

n: 890

Globals

WATCH

CALL STACK

Paused on breakpoint

fact	example.py	10:1
<lambda>	example.py	13:1
trampoline	example.py	5:1
<module>	example.py	15:1

Users > tommymoawad > Desktop > example.py > fact

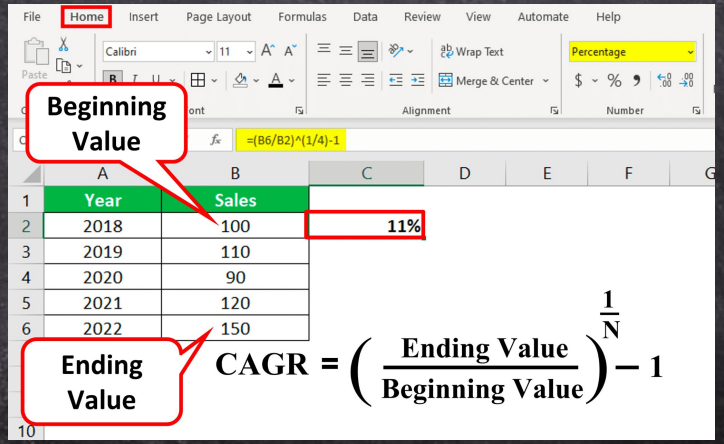
```python
# Trampoline: A loop that goes from “thunk” to “thunk”
# Thunk: A function that takes no args and invokes another function
def trampoline(x):
    while callable(x):
        x = x()   # ← this is a “thunk”
    return x


def fact(acc, n):
    print(acc, 1)
    if n == 1:
        return acc
    return lambda: fact(acc * n, n - 1)

print(trampoline(fact(1, 1000)))
```

Practical Use Cases

1. Implement naturally recursive formulas without major refactors
2. Anything that uses a time series
   a. Finance
   b. Physics



Practical Use Cases

Naturally recursive functions

```python
def binary_search(nums, key):
    left_idx, right_idx = 0, len(nums)
    while right_idx > left_idx:
        middle_idx = (left_idx + right_idx) // 2
        if nums[middle_idx] > key:
            right_idx = middle_idx
        elif nums[middle_idx] < key:
            left_idx = middle_idx + 1
        else:
            return middle_idx
    return None
```

Time series (Finance)