Week 11

Memory Model revisited, Hashing

Agenda

- Memory model revisited
  - For nested function calls
  - For list comprehension
  - For Class Hierarchy
  - For Mutable Data types

Memory model revisited

Call Stack

Created for Function Calls, Nested function calls, Recursive function calls

Heap

Created for all objects

Protected for each function execution “environment”

Call Stack

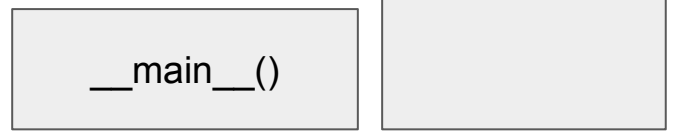
```
def f(n: int) -> int:
    """
    Return g(n + 1)
    """
    x = 1
    x = g(n + 1)
    return x


def g(n: int) -> int:
    """
    Return h(n + 1)
    """
    x = 2
    x = h(n + 1)
    return x


def h(n: int) -> int:
    """ return n + 1
    """
    x = n + 1
    return x
```

```
if __name__=="__main__":

    # tracing nested function calls
    x = 7
    f(1)
```

Call Stack

```
def f(n: int) -> int:
    """
    Return g(n + 1)
    """
    x = 1
    x = g(n + 1)
    return x


def g(n: int) -> int:
    """
    Return h(n + 1)
    """
    x = 2
    x = h(n + 1)
    return x


def h(n: int) -> int:
    """ return n + 1
    """
    x = n + 1
    return x
```

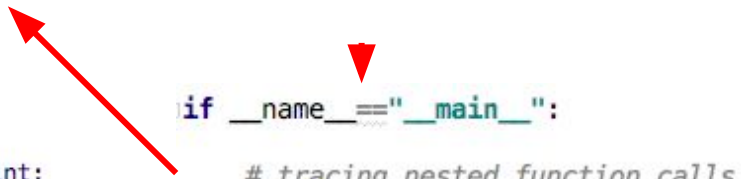
```
if __name__=="__main__":

    # tracing nested function calls
    x = 7
    f(1)
```

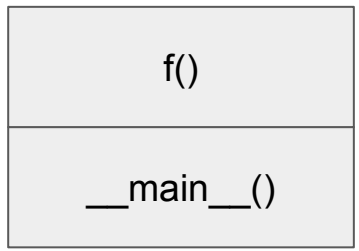


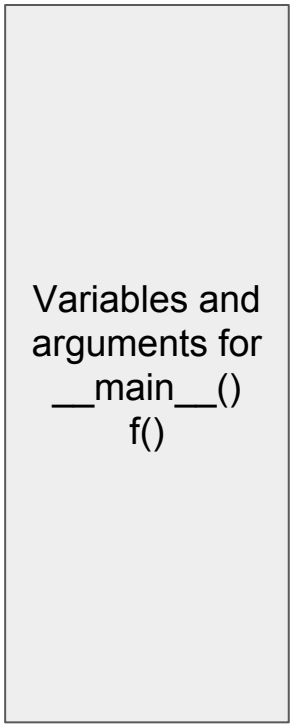
Call Stack

```
def f(n: int) -> int:
    """
    Return g(n + 1)
    """
    x = 1
    x = g(n + 1)
    return x


def g(n: int) -> int:
    """
    Return h(n + 1)
    """
    x = 2
    x = h(n + 1)
    return x


def h(n: int) -> int:
    """ return n + 1
    """
    x = n + 1
    return x
```

```
if __name__=="__main__":

    # tracing nested function calls
    x = 7
    f(1)
```

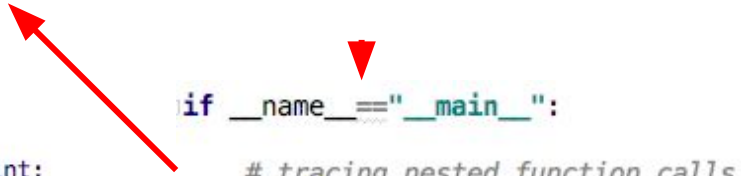




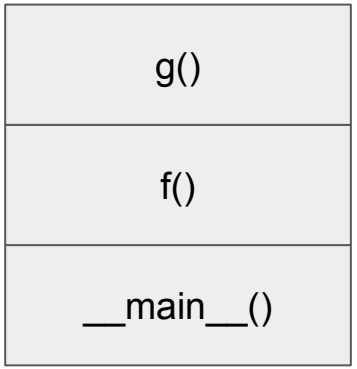


Call Stack

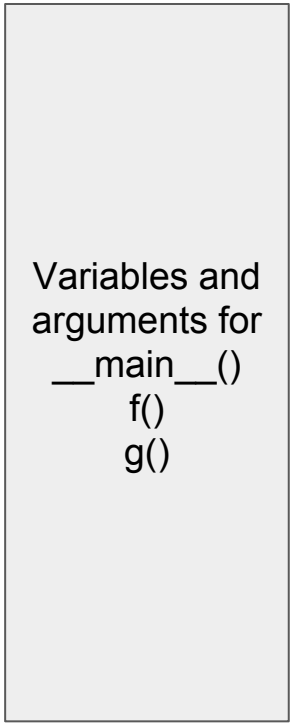
```
def f(n: int) -> int:
    """
    Return g(n + 1)
    """
    x = 1
    x = g(n + 1)
    return x

def g(n: int) -> int:
    """
    Return h(n + 1)
    """
    x = 2
    x = h(n + 1)
    return x

def h(n: int) -> int:
    """ return n + 1
    """
    x = n + 1
    return x

if __name__=="__main__":
    # tracing nested function calls
    x = 7
    f(1)
```



Call Stack

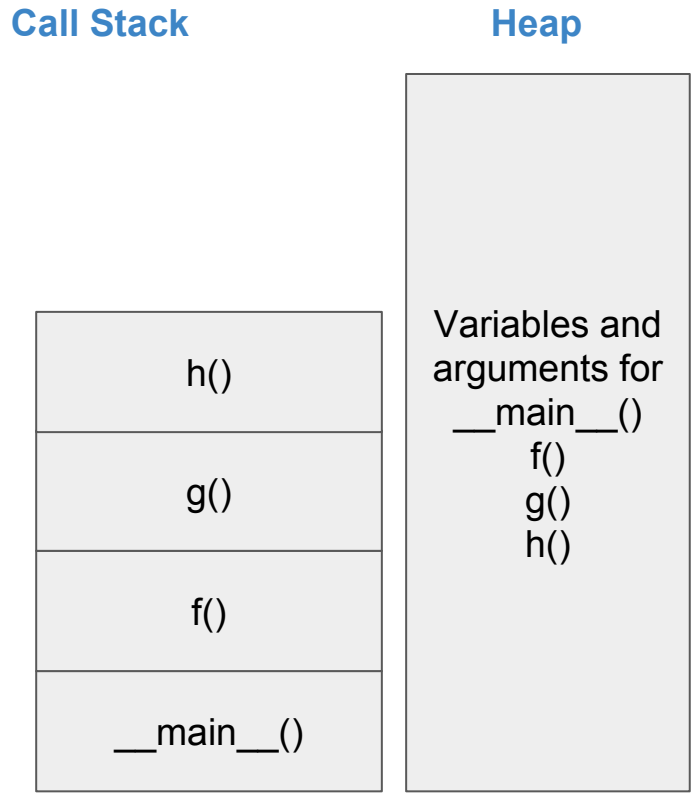


Heap



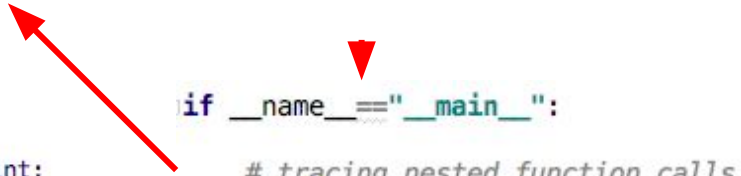
Call Stack

```
def f(n: int) -> int:
    """
    Return g(n + 1)
    """
    x = 1
    x = g(n + 1)
    return x

def g(n: int) -> int:
    """
    Return h(n + 1)
    """
    x = 2
    x = h(n + 1)
    return x

def h(n: int) -> int:
    """ return n + 1
    """
    x = n + 1
    return x

if __name__=="__main__":
    # tracing nested function calls
    x = 7
    f(1)
```

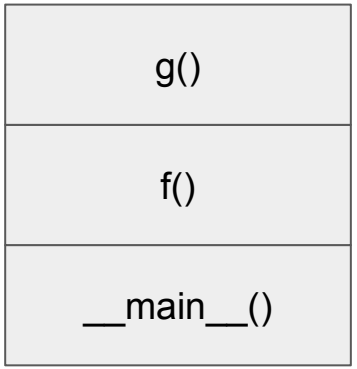


Call Stack

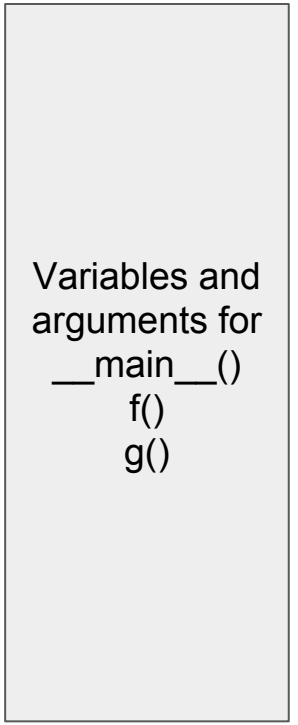
```
def f(n: int) -> int:
    """
    Return g(n + 1)
    """
    x = 1
    x = g(n + 1)
    return x

def g(n: int) -> int:
    """
    Return h(n + 1)
    """
    x = 2
    x = h(n + 1)
    return x

def h(n: int) -> int:
    """ return n + 1
    """
    x = n + 1
    return x

if __name__=="__main__":
    # tracing nested function calls
    x = 7
    f(1)
```



Call Stack



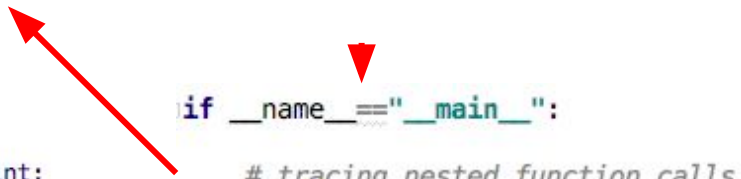
Heap

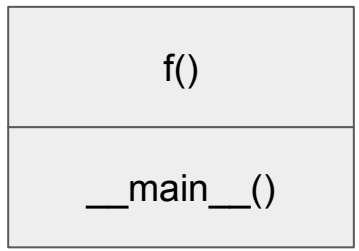


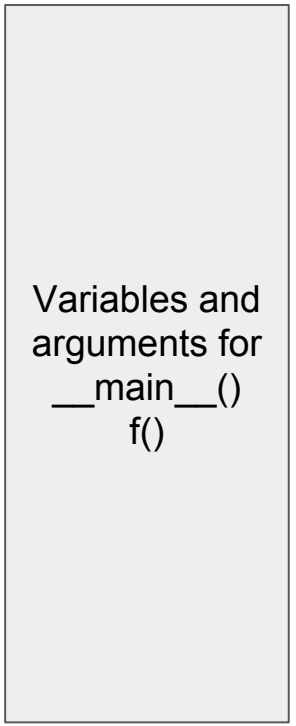
Call Stack

```
def f(n: int) -> int:
    """
    Return g(n + 1)
    """
    x = 1
    x = g(n + 1)
    return x


def g(n: int) -> int:
    """
    Return h(n + 1)
    """
    x = 2
    x = h(n + 1)
    return x


def h(n: int) -> int:
    """ return n + 1
    """
    x = n + 1
    return x
```

```
if __name__=="__main__":

    # tracing nested function calls
    x = 7
    f(1)
```



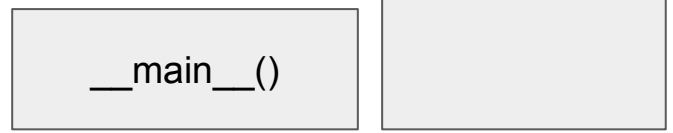




Call Stack

```
def f(n: int) -> int:
    """
    Return g(n + 1)
    """
    x = 1
    x = g(n + 1)
    return x


def g(n: int) -> int:
    """
    Return h(n + 1)
    """
    x = 2
    x = h(n + 1)
    return x


def h(n: int) -> int:
    """ return n + 1
    """
    x = n + 1
    return x
```

```
if __name__=="__main__":

    # tracing nested function calls
    x = 7
    f(1)
```



Call Stack

Call Stack

Heap

```
def f(n: int) -> int:
    """
    Return g(n + 1)
    """
    x = 1
    x = g(n + 1)
    return x


def g(n: int) -> int:
    """
    Return h(n + 1)
    """
    x = 2
    x = h(n + 1)
    return x


def h(n: int) -> int:
    """ return n + 1
    """
    x = n + 1
    return x
```

```
if __name__=="__main__":

    # tracing nested function calls
    x = 7
    f(1)
```

Memory Model for Nested Calls

```
def f(n: int) -> int:
    """
    Return g(n + 1)
    """
    x = 1
    x = g(n + 1)
    return x

def g(n: int) -> int:
    """
    Return h(n + 1)
    """
    x = 2
    x = h(n + 1)
    return x

def h(n: int) -> int:
    """ return n + 1
    """
    x = n + 1
    return x
```

```
if __name__=="__main__":

    # tracing nested function calls
    x = 7
    f(1)
```

Key Insight

1. Every function has its *own* execution environment in the call stack
2. All the local variables and arguments are the function’s property
3. Unique copies for two variables with the same name in two functions
4. All local variables are gone at the end of function execution

Tracing multiple function definitions

```
def h(n: int) -> int:
    """ return n + 1
    """
    x = n + 1
    return x
```

```
def h(n: int) -> int:
    """ return n + 1
    """
    x = n + 2
    return x
```

Tracing comprehension

```
def comprehension(list_:list)->list:
    list_=[x**2 for x in list_]
    return list_
```

Tracing Recursion

```
def sum_list(list_: Union[list, int]) -> int:
    """
    Return list_, if it is an int, or the sum of all elements
    of list_ and its sub-lists.
    """
    if not isinstance(list_, list):
        return list_
    else:
        flat_list = [sum_list(x) for x in list_]
        return sum(flat_list)
```

Tracing Class Hierarchy

```
class Parent:
    """
    Parent class...
    """
    def f(self, n: int) -> int:
        """
        Return 2 * g(n)
        """
        return 2 * self.g(n)

    def g(self, n) -> int:
        """
        Return 3 * n
        """
        return 3 * n
```

```
class Child(Parent):
    """
    Child class...
    """
    def g(self, n: int) -> int:
        """
        Return n

        Overrides Parent.g
        """
        return n
```

```
child = Child()
child.f(1)
```

Exercise: Guess the output

```
def filter_list(list_: list, predicate: Callable[[object], bool]) -> None:
    """
    Modify list_ so that it contains only elements that satisfy predicate.
    """
    temp_list = []
    while len(list_) > 0:
        if predicate(list_[0]):
            temp_list.append(list_.pop(0))
        else:
            scrap = list_.pop(0)
    list_ = temp_list
    # placeholder
    x = 'blah'


if __name__=="__main__":
    list_ = [1, 4, 6, 8]
    def pred(n): return n > 5
    filter_list(list_, pred)
    print(list_)
```