EECS 390 – Lecture 8

Higher-Order Functions

2/6/24

Agenda

- Function Objects
- Functions as Parameters
- Nested Functions

Function Objects and State

- A ***function object*** (also called a ***functor***) is an object that isn't a function but provides the same interface

- Allowing the function-call operator to be overloaded enables function objects to be defined

- Function objects can have state that is associated with an instance of the functor
  - State shared among all invocations of the same instance
  - Different than top-level functions, which only have state that is associated with a single invocation or with all invocations of the function

Function Objects in C++

- Functors can be written by defining a class that overloads the `operator()` member function

```
class Counter {
public:
  Counter : count(0) {}
  int operator()() {
    return count++;
  }
private:
  int count;
};
```

Can have parameters, just like functions

```
Counter counter1, counter2;
cout << counter1() << endl; // prints 0
cout << counter1() << endl; // prints 1
cout << counter1() << endl; // prints 2
cout << counter2() << endl; // prints 0
cout << counter2() << endl; // prints 1
cout << counter1() << endl; // prints 3
```

Function Objects in Python

- Functors overload the `__call__` special method

```
class Counter:
    def __init__(self):
        self.count = 0
    def __call__(self):
        self.count += 1
        return self.count - 1
```

More parameters can go here

```
counter1 = Counter()
counter2 = Counter()
print(counter1()) # prints 0
print(counter1()) # prints 1
print(counter1()) # prints 2
print(counter2()) # prints 0
print(counter2()) # prints 1
print(counter1()) # prints 3
```

Function Pointers

- C and C++ allow top-level functions to be passed by pointer

```
void apply(int *A, size_t size, int (*f)(int)) {
  for (; size > 0; --size, ++A)
    *A = f(*A);
}

int add_one(int x) {
  return x + 1;
}

int main() {
  int A[5] = { 1, 2, 3, 4, 5 };
  apply(A, 5, add_one);
  cout << A[0] << ", " << A[1] << ", " << A[2]
       << ", " << A[3] << ", " << A[4] << endl;
}
```

Automatically converted to function pointer

Environment of Use

- A function passed as a parameter has three environments that can be associated with it
  - The environment where it was defined
  - The environment where it was referenced
  - The environment where it was called
- Scope policy determines which names are visible in the function
  - Static/lexical scope: names visible at the definition point
  - Dynamic scope: names visible at the point of use
- In dynamic scope, point of use can be where a function is referenced or where it is called

Binding Policy

- ***Shallow binding***: non-local environment is environment from where a function is called
- ***Deep binding***: non-local environment is environment from where a function is referenced

```
int foo(int (*bar)()) {
  int x = 3;
  return bar();
}

int baz() {
  return x;
}

int main() {
  int x = 4;
  print(foo(baz));
}
```

Non-local environment in shallow binding

Non-local environment in deep binding

C-like code with dynamic scope

Evaluating Function Calls

- Determine non-local environment
  - Static scope: active environment when the function is defined
  - Dynamic scope with deep binding: active environment when the function is named
  - Dynamic scope with shallow binding: active environment when the function is called
- Create new activation record and pass parameters
  - Call by value: obtain r-value of argument and copy it into the new activation record
  - Call by reference: obtain l-value of argument and bind the parameter to the corresponding object
  - Call by result: obtain l-value of argument, create uninitialized storage in new activation record
  - Call by name: create thunk from argument expression and current environment

Evaluating Function Calls

- Pause caller, execute body of callee in environment consisting of new activation record and the function's non-local environment
- When callee returns:
  - Store return value (usually in activation record of caller)
  - Copy r-values of call-by-result parameters into objects associated with argument l-values
  - Destroy activation record of callee (if using stack-based memory management)
  - Resume execution of caller
- The evaluation result of the function call is the return value of the callee

Nested Functions and Closures

- The ability to create a function from within another function is a key feature of functional programming
- Static scope requires that the newly created function have access to its definition environment
- A ***closure*** is the combination of a function and its enclosing environment
- Variables from the enclosing environment that are used in the function are ***captured*** by the closure

Nested Functions and State

- A closure encompasses state that can be accessed by the newly created function

```
def make_greater_than(threshold):
    def greater_than(x):
        return x > threshold
    return greater_than
```

threshold captured from non-local environment

```
>>> gt3 = make_greater_than(3)
>>> gt30 = make_greater_than(30)
>>> gt3(2)
False
>>> gt3(20)
True
>>> gt30(20), gt30(200)
(False, True)
```

Modifying Non-Local State

- Languages may allow non-local variables to be modified

```
def make_account(balance):
    def deposit(amount):
        nonlocal balance
        balance += amount
        return balance
    def withdraw(amount):
        nonlocal balance
        if 0 <= amount <= balance:
            balance -= amount
            return amount
        else:
            return 0
    return deposit, withdraw
```

```
>>> deposit, withdraw = \
        make_account(100)
>>> withdraw(10)
10
>>> deposit(0)
90
>>> withdraw(20)
20
>>> deposit(0)
70
>>> deposit(10)
80
>>> withdraw(100)
0
>>> deposit(0)
80
```

We will come back to data abstraction using functions later

Decorators

- A common pattern in Python is to transform a function or class by applying a higher-order function to it, called a ***decorator***

- Standard syntax for decorating functions:

```
@<decorator>
def <name>(<parameters>):
    <body>
```

- Mostly equivalent to:

```
def <name>(<parameters>):
    <body>

<name> = <decorator>(<name>)
```

Trace Example

- Example: decorator that traces function calls

"Representation" string

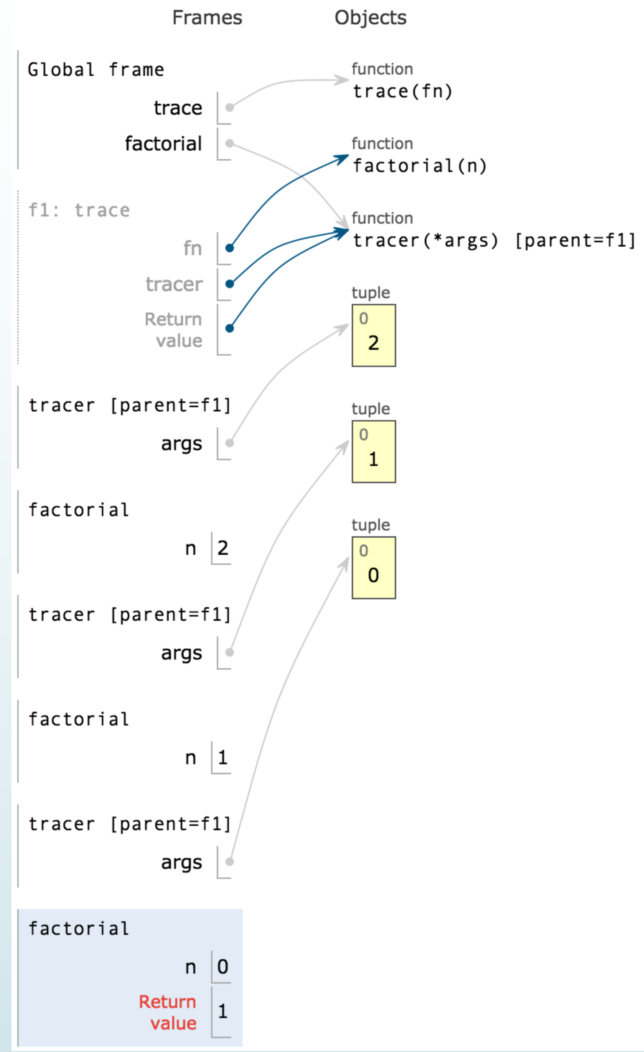
```
def trace(fn):
    def tracer(*args):
        arg_str = ', '.join(repr(arg)
                                for arg in args)
        print(f'{fn.__name__}({arg_str})')
        return fn(*args)
    return tracer

@trace
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

```
>>> factorial(5)
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
factorial(0)
120
```

Mutual Recursion

- A decorated recursive function results in ***mutual recursion*** where multiple functions make recursive calls indirectly through each other

```
>>> factorial(2)
factorial(2)
factorial(1)
factorial(0)
2
```



This example on Python Tutor: https://goo.gl/issW90

Partial Application

- Specify some arguments to a function, then specify remaining arguments later
- If function takes n arguments and k are supplied, results in function that takes $n - k$ arguments

```
def partial(func, *args):
    def newfunc(*nargs):
        return func(*args, *nargs)
    return newfunc

>>> power_of_two = partial(pow, 2)
>>> power_of_two(3)
8
>>> power_of_two(7)
128
```

C++ has `bind()` template to do this in `<functional>`

Currying

- Transforms a function that takes n arguments into a series of n functions that each take in one argument

- In some languages, all functions are curried

```
def curry2(func):
    def curriedA(a):
        def curriedB(b):
            return func(a, b)
        return curriedB
    return curriedA

>>> curried_pow = curry2(pow)
>>> curried_pow(2)(3)
8
```

Uncurrying

- We can also do the reverse transformation

```
def uncurry2(func):
    def uncurried(a, b):
        return func(a)(b)
    return uncurried

>>> uncurried_pow = uncurry2(curried_pow)
>>> uncurried_pow(2, 3)
8
```