Higher-Order Functions

Announcements

Office Hours: You Should Go!

You are not alone!

http://cs61a.org/office-hours.html

Iteration Example

The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987

The next Fibonacci number is the sum of the current one and its predecessor.

Go Bears!

Designing Functions
Describing Functions

A function’s **domain** is the set of all inputs it might possibly take as arguments.

A function’s **range** is the set of output values it might possibly return.

A pure function’s **behavior** is the relationship it creates between input and output.

```python
def square(x):
    """Return X * X."""
    x is a number
    square returns a non-negative real number
    square returns the square of x
```

A Guide to Designing Function

Give each function exactly one job, but make it apply to many related situations

Don’t repeat yourself (DRY): Implement a process just once, but execute it many times

```python
>>> round(1.23, 1)
1.2
>>> round(1.23, 0)
1
>>> round(1.23, 5)
1.23
>>> round(1.23)
1
```

Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Finding common structure allows for shared implementation

Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

```python
\[ \sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 = 15 \]
\\
\[ \sum_{k=1}^{3} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225 \]
\\
\[ \sum_{k=0}^{4} \frac{8}{(4k - 1)(4k - 3)} = \frac{8}{3} + \frac{8}{5} + \frac{8}{9} + \frac{8}{175} + \frac{8}{23} = 3.04 \]
```

Summation Example

```python
def cube(k):
    """Function of a single argument (not called "term")"
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence."

>>> summation(5, cube)
225

The cube function is passed as an argument value

while k <= n:
    total, k = total + term(k), k + 1
return total

The function bound to term gets called here
```

Functions as Return Values

```python

def make_adder(n):
    """Return a function that takes one argument k and returns k + n."

>>> add_three = make_adder(3)
>>> add_three(4)
7
```

```
def compose1(f, g):
    """Return a function that composes f and g."

    f, g
    functions of a single argument

    """
    def h(x):
        return f(g(x))
    return h
```

```
@main
def run():
    interact()
```

Demo
Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame.

```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n."
    def adder(k):
        return k + n
    return adder
```

Call Expressions as Operator Expressions

An expression that evaluates to a function

```
def square(x):
    return x * x
```

An expression that evaluates to its argument

```
def sum(term, n):
    total = 0
    k = 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

Lambda Expressions

```
def adder(k):
    return k
```

```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n."
    def adder(k):
        return k + n
    return adder
```

Lambda Expressions Versus Def Statements

```
square = lambda x: x * x
```

```
def square(x):
    return x * x
```

Lambda expressions are not common in Python, but important in general.
Lambda expressions in Python cannot contain statements at all!

Return Statements

A return statement completes the evaluation of a call expression and provides its value:

```
def end(n, d):
    """Print the final digits of N in reverse order until D is found."
    while n > 0:
        last, n = n % 10, n // 10
        print(last)
        if d == last:
            return None
```

Return
If Statements and Call Expressions

Let's try to write a function that does the same thing as an if statement.

```
def if_(c, t, f):
    if c:
        return t
    else:
        return f
```

Control Expressions

Logical Operators

To evaluate the expression `<left> and <right>`:
1. Evaluate the subexpression `<left>`.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression `<right>`.

To evaluate the expression `<left> or <right>`:
1. Evaluate the subexpression `<left>`.
2. If the result is a true value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression `<right>`.

```
>>> x = 0
>>> abs(1/x if x != 0 else 0)
0
```

Conditional Expressions

A conditional expression has the form

```
<consequent> if <predicate> else <alternative>
```

Evaluation rule:
1. Evaluate the `<predicate>` expression.
2. If it's a true value, the value of the whole expression is the value of the `<consequent>`.
3. Otherwise, the value of the whole expression is the value of the `<alternative>`.