CISS450: Artificial Intelligence
Lecture 12: More on Functions

Yihsiang Liow

Agenda

- More argument tricks. We will see how Python support arbitrarily many arguments for functions

Arbitrarily Many Non-kw Arguments

- The following are some interesting features of Python function arguments
- Try:

```
def f(*x):
    print(type(x), x)
f()
f(1)
f(1,2)
f(1,2,3)
```

- What do you think `*argument` mean in Python argument passing?

Arbitrarily Many Non-kw Arguments

- Complete and test the following:

```
def avg(*x):
    """Returns the average"""

    return a
```

Arbitrarily Many Kw Arguments

- Try:

```
def f(**x):
    print type(x), x


f()
f(x=1)
f(x=1,y=2)
f(x=1,y=2,z=3)
```

- What do you think `**argument` mean in Python's argument passing?

Argument Matching

- If you have f(x=0, **y), and you call f(x=1) does that mean the first argument x is set to 1, or does it mean y={'x':1}? We need rules for matching arguments passed in with the parameters in the function declaration.
- Order of specifying parameters:
  - Non-keyword arguments
  - Keyword arguments
  - One *argument
  - One **argument

Arbitrarily Many Arguments

- Example: Which of the following are correct? Incorrect? (Why?)

```
def f0(x,y=0,*z,**w): pass
def f1(x=0,y): pass
def f2(x,y=0,*x): pass
def f3(y=0,**w): pass
def f4(y=0,**a,*b): pass
def f5(**a,**b): pass
```

Arbitrarily Many Arguments

- Order of parameter matching:
  - Match non-keyword arguments by position
  - Match keyword arguments by keywords
  - Assign non-keyword arguments to *argument
  - Assign keyword arguments to **argument
- Example:

```
def f(x0,x1,x2,x3=3,x4=4,x5=5,*x6,**x7):
    print(x0,x1,x2,x3,x4,x5,x6,x7)
f('a','b','c','d','e','f','g','h','i')
f('a','b','c','d','e','f','g','h','i',j='j',k='k',l='l')
f(x5='a',x4='b',x3='c',x2='d',x1='e',x0='f',j='j',k='k')
```

Evaluation of Default Arguments

- Here's an ***important warning***. Test the following code:

```
def f(x, xs=[]):
    xs.append(x)
    return xs


print(f(1))
print(f(2))
print(f(3))
```

- What do you think is happening?

Evaluation of Default Arguments

- Try this instead:

```
def f(x, xs=None):
    if xs==None:
        xs = []
    xs.append(x)
    return xs

print(f(1))
print(f(2))
print(f(3))
```

- Compare this version of f with the previous. Make sure you understand the difference.

Function as first class value

- The following concepts are part of “functional programming”: function are like any values (int, boolean, string, etc.)
- Functions can be passed into functions:

```
def f(x): return x + 1
def g(x): return x(42)
print(g(f))
```

Function as first class value

- Function can be returned from function:

```
def f(x):
    def g(y): return x + y
    return g
print(f(5)(7))
```

Polymorphism

- In C++ you do function overloading:

```
int f(int x) { return x + 1; }

double f(double x) { return x *
3.14; }
```

- In Python you can do this for overloading:

```
def f(x):
    if isinstance(x, int):
        return x + 1
    elif isinstance(x, double):
        return x * 3.14;
```

Polymorphism

- The second argument of isinstance can be a tuple of types:

```
def addone(x):
    if isinstance(x, (int, float)):
        return x + 1
    elif isinstance(x, list):
        return [y + 1 for y in x]
print(addone(42))
print(addone([2, 3, 5, 7, 11]))
```

Examples

```
def ifelse(b, x, y):
    if b: return x
    else: return y
def nextprime(x):
    # returns the prime is that > x
    def isprime(x):
        for d in range(2, x):
            if x % d == 0:
                return False
        return True
    if isprime(x + 1):
        return x + 1
    else:
        return nextprime(x + 1)
p = 1
for _ in range(10):
    p = nextprime(p); print(p)
```

Examples

```
def avg(f, a, b):
    return (f(b) – f(a)) / float(b – a)
def compose(f, g):
    def h(x): return f(g(x))
    return h
def product(xs, ys):
    p = []
    for x in xs:
        for y in ys:
            p.append((x, y))
    return p
```

Coda

- It takes time to learn how to use the rich variety of argument passing in Python. But once you're familiar with it, you'll love it.
- Create your own experiments and learn to use it.
- When we have time, we will come back to functions again and look at ***functional programming*** features of Python.