Review:
Evaluation in Python

15-110 – Friday 09/25

Learning Goals

- Difference between program and data in Python

- Review of how evaluation works in Python

Program vs. Data in Python

Program

Statements:

- assignment statements: x=5
- **if/elif/else** statements
- **for** and **while** loops
- function definitions with **def**
- **return** statements

Variables: x, foo

Expressions: f(3) + 5*g(2,x)

Data

Python objects:

- ints
- floats
- strings
- booleans
- None
- other types of objects

Can be stored in a variable.

Evaluation in Python

Evaluation converts an **expression** (program stuff) to a **value** (data stuff).

Evaluation is a key part of how Python runs your program.

How does it work? By following a set of **evaluation rules**.

Evaluation Rule for Constants

Ints, floats, strings, bools, and None are **constants** in Python.

A constant always evaluates to itself.

```
3 ⇒ 3
True ⇒ True
"banana" ⇒ "banana"
```

Evaluation Rule for Variables

A variable evaluates to the value stored in that variable.

Assume we have executed the statement `x = 5`.

Afterwards, we have:

x ⇒ 5

Note that x is a variable (program stuff), while 5 is a value (data stuff).

Evaluation Rule for Function Call Expressions

If we have an expression of form $e_0(e_1, \ldots, e_n)$ where the e_i are expressions:

1. **Eval step:** Evaluate each expression e_i to get its value v_i
2. v_0 must be a function object
3. **Apply step:** Apply function object v_0 to the evaluated args. $v_1 \ldots v_n$
4. The object returned by the function is the value of the expression

Example: Evaluate max(3,x)

Assume we've set `x=5`.

Eval: `max(3,x)`

  Eval: `max` ⇒ `<built-in function max>`

  Eval: `3` ⇒ `3` *evaluation rule for constants*

  Eval: `x` ⇒ `5` *evaluation rule for variables*

  Apply `<built-in function max>` to arguments `3` and `5`

  Return value from max is `5`

`max(3,x)` ⇒ `5`

Evaluation Rule for Operators

Operators are syntactic shorthand for method calls.

`x+7` is Python shorthand for `x.__add__(7)`

Methods are functions tied to a specific type, e.g., int.

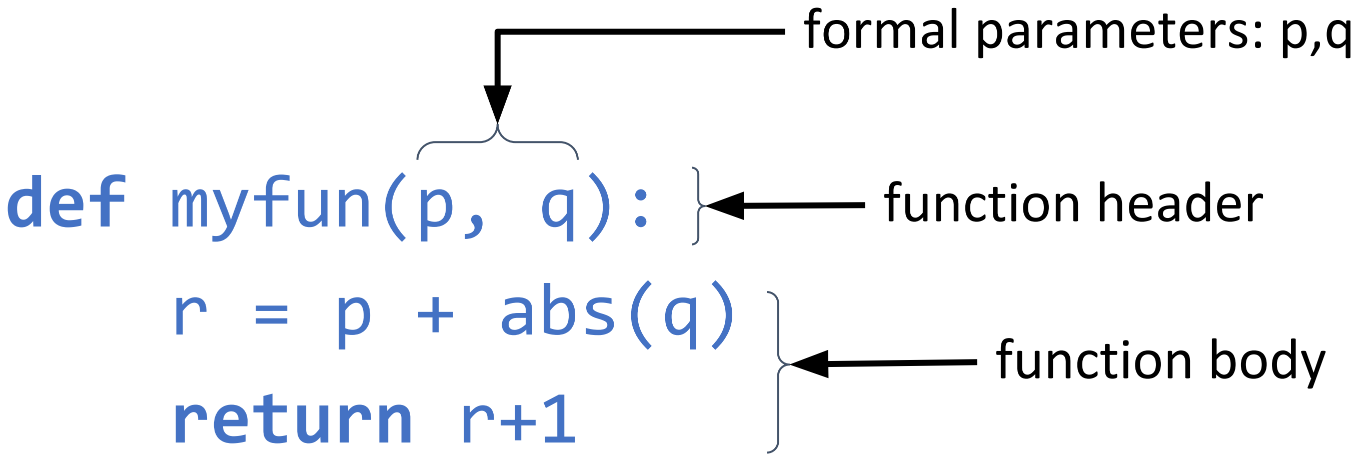
Evaluate operator expressions the same way as function calls.

Example: Evaluate x + y * 3

Note that `x+y*3` is shorthand for `x.__add__(y.__mul__(3))`

```
Eval: x + y * 3
    Eval: x  ⇒  5
    Eval: y * 3
        Eval: y  ⇒  4
        Eval: 3  ⇒  3
        Apply __mul__ to arguments 4 and 3
        Return value of __mul__  is 12
    y * 3  ⇒  12
    Apply __add__ to arguments 5 and 12
    Return value of __add__ is 17
x + y * 3  ⇒  17
```

Components of a User-Defined Function



formal parameters: p,q

function header

function body

```
def myfun(p, q):
    r = p + abs(q)
    return r+1
```

```
myfun(5, -8)
```

Applying a User-Defined Function

If we have a function call $e_0(e_1, \ldots, e_n)$ where the e_i are expressions:

1. **Eval step:** Evaluate each e_i to get its value v_i

2. v_0 must be a function object with formal parameters $p_1 \ldots p_n$

3. **Apply step:**
   a. Create a new call frame on the call stack with local variables $p_1 \ldots p_n$
   b. Assign $p_i = v_i$ for i from 1 to n
   c. Execute the statements in the function body, one at a time
   d. If a **return** statement is executed, stop: use the return value as the result of the function call. Otherwise, if we reach the end of the function body, use None as the result of the function call.
   e. Pop the call frame off the call stack and return the result

Evaluating myfun(5, -8)

```
Eval: myfun(5, -8)
    Eval: myfun ⇒ <function myfun>
    Eval: 5 ⇒ 5
    Eval: -8 ⇒ -8
    Apply <function myfun> to inputs 5 and -8
        Create new call frame with p=5 and q=-8
        Execute statement: r = p + abs(q)
            Eval: p+abs(q) ⇒ 13
            Set local variable r to 13
        Execute statement: return r+1
            Eval: r+1 ⇒ 14
            Return value is 14
        Pop the call frame
myfun(5, -8) ⇒ 14
```

Executing Statements

To **execute** a statement means to perform the action associated with that statement type.

Each statement type has its own rule for how to execute it.

Assignment Statement

Syntax: `var = expr`

`var` must be a variable name

`expr` must be an expression

Execution rule for assignment statements:

1. Evaluate `expr` to get a value
2. Store that value in `var`

if/else Statements

Syntax:

```
if expr:
    body1
else:
    body2
```

Execution rule for if/else statements:

1. Evaluate expr
2. If the result is True, execute the statements in body1
3. Otherwise execute the statements in body2

for Statements

Syntax:

```
    for var in expr:
        body
```

Execution rule for `for` statements (assuming `expr` is `range(e1,e2,e3)`):

1. Evaluate `expr`:
    a. Evaluate `range` to get a function object
    b. Evaluate expressions `e1`, `e2`, `e3` to get values v1, v2, v3
    c. Call the range function object on the values v1, v2, v3 to get a <u>range object</u>
2. While the range object still has values left to produce:
    a. Assign the next value produced by the range object to `var`
    b. Execute the statements in `body`
3. When the range object has run out of values, the for loop is complete.

Expressions Can be Statements

The body of a function is a collection of statements.

Some of these statements can just be expressions, like x+5.

Execution rule for expressions as statements:

1. Evaluate the expression.
2. Throw the result away.

Why have expressions be statements? Side effects!

```
print("x is", x)
```

Learning Goals

- Difference between program and data in Python

- Review of how evaluation works in Python

- **Feedback: https://bit.ly/110-feedback**