CS61A Lecture 3

Saturday, August 31st 2019

Special video-only lecture

- How rules of evaluation in Python can lead to interesting behavior. Questions like:
  - What happens if I use a name twice in two different contexts?
- Looking at a lot of environment diagrams
  - The program that generates them is new: code.cs61a.org
  - Click the bug button in the top-right corner to use the new environment diagram
  - tutor.cs61a.org is still available with the old environment.
- Hog project will be released Monday and due on Thursday, September 12th.
  - First part is individual, the rest can be done with a partner.

Print and None

If you type an expression into the interpreter, ti will then display the value of the expression.

The same case occurs when you call `print`.

```
>>> -2
2
>>> print(-2)
-2
```

Two different things have happened here, even though it looks the same.

```
>>> 'Go Bears!'
'Go Bears!'
>>> print('Go Bears!')
Go Bears!
```

You can see one difference here, as the print has no quotation marks, while directly evaluating shows the string value reproduced directly from the literal.

Another difference occurs when you type `None`, where nothing will occur:

```
>>> None
```

Meanwhile,

```
>>> print(None)
None
```

Python has rules for automatically displaying the value of anything you type in. `None` is a special case where nothing gets displayed automaticallu, but printing it can make it automatically appear.

Print can also show values separated by spaces.

Here's an interesting case. What happens when you `print(print(<something>))` ?

```
>>> print(print(1),print(2))
1
2
None None
```

Let's understand what happened.

None indicates that nothing has been returned by a function in Python.

> A function that does not explicitly return a value will return `None` .

`None` is not displayed automatically by the interprter as the value of an expression.

Take this example:

```
>>> def does_not_square(x):
        x*x
```

This function does not have a return value.

```
>>> does_not_square(4)
```

The value `None` is not displayed according to the third rule.

```
>>> sixteen = does_not_square(4)
```

The name `sixteen` is now bound to the value `None` .

```
>>> sixteen + 4
```

This does not return 20 like it would if you'd properly written the function, but instead a `TypeError` , because you've tried to add a `None` to an `int` . In other words, you've tried to add something to nothing.

This is because there are two kinds of functions, **pure functions** and **non-pure functions**.

Pure functions are functions like `abs` , which is in essence a closed pipe that only outputs the return value. It has no side effects.

Non-pure functions are functions like `print` . The pipe that represents a function with an output and an input also has a leak, which is the side effect. In `print` 's case, it takes an input, displays whatever is between the brackets, then returns `None` . In other words, it did something other than return an output.

A side effect isn't a value, it's anything that happens as a consequence of calling a function. This is the cause behind the interesting behavior when `print` is nested.

Let's return to the original example of:

```
print(print(1),print(2))
```

According to the rules, we first evaluate the operator, which is `print`.

The first operand is another call expression, which we evaluate by evaluating its operator and operand. When we call `print` on 1, we have a side effect of displaying the number 1. While evaluating this operand, we displayed the number 1, but this 1 is not the value of anything. We did get a value for this expression, but the value was `None`. Next, we do the same thing to the right. So now, we pass in `None` `None` as the input to the parent `print`, and this displays `None` `None`, and it itself returns a value of `None`, which doesn't get displayed.

Multiple Environments

When Python evaluates a program, different expressions can be evaluated in different environments. There can be multiple environments in the same environment diagram. We're gonna look at one of the examples showed last time in detail, to understand what exactly is going on.

Life Cycle of a User-Defined Function

- `def` statements creates functions, spans multiple lines, has a name, formal parameters separated by commas, and a body identified by the indented lines after the first.
  - **What happens?** A new function is created, and the name is bound to that function *in the current frame*. None of the computation in the function is done, this only happens when the function is called.
- Call expressions are when functions are called by their name. The operands are the values in the parentheses.
  - **What happens?** The operators and operands are evaluated. The function is called on the arguments.
- Calling/applying the function on the operands:
  - **What happens?** A new frame is created, in which the formal parameters are bound to the arguments we are passing in, and the body is executed *in that new environment*.

Take the following example.

```
square(square(3))
```

The environment diagram will first show that two new frames are created, `f1` and `f2`, in which they both have the variable `x` and a return value, but the first `x` is 3 and the second is `9`.

> **Why isn't the parent of `f2` `f1`, when `f2` is called within `f1`?**
> The parent frame is the frame in which the function is defined. `square` is defined in the global frame, therefore the new frames created by calling it will have the global frame as their parents.

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

In the above example, there are three environments:

1. The global frame alone

2. The f1 frame, followed by the global frame
3. The f2 frame, followed by the global frame

In this case, there are three different environments, and none that include all three, but one environment per frame.

If you start with a particular frame, you can always find the whole environment, just by following the parents of the frames.

Let's say we are interested in the frame f2, so the next frame is its parent, the global frame. Global frames are always the last frame because they never have any parents.

> Names have no meaning without these environments; these are the things that endow `mul`, `x` and whatever names with some sort of meaning.

Every expression is evaluated in the context of an environment, which allows us to figure out what names mean.

> A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

In `square(square(3))`, you will find two names, `mul` and `x`. We first look for `x` in the current environment, and we find it in the first frame, which is the earliest frame of the environment in which the name is found. When we look up `mul`, we first look in `f2`, where `mul` isn't there, and then we look in the global frame, where `mul` is, bound to the function that multiplies.

Names have different meanings in different environments, because each frame can have different meanings for the same name. A call expression and the body of the function being called are evaluated in different environments. For example,

```
def square(square):
    return square*square
square(4)
```

does in fact return 16, because the name `square` and the function `square` are evaluated in different environments. `square(...)` is evaluated in the global frame, while `(square)` is evaluated in the `f1` frame, where there is a different binding for `square`, and thus the binding in the global frame is never encountered.

Miscalleneous Python Features

How Operators Work

Operators such as + or * haven't really been discussed. How they really work is complicated and will be saved for later in the course, so for now, think of them as short hand for calling built-in functions such as `add` and `mul`.

Let's now talk about division. There are two kinds of division:

- True division: `2019/10` gives `201.9`

- Integer division gives only the number of times the divisor goes into the quotient: `2019//10` gives `201`, not including any remainer

The corresponding functions are `truediv` and `floordiv`.

We can get the remainder with the mod operator, `%` or the `mod` operator.

Why would we want these functions? Because `floordiv` and `mod` are exact, while `truediv` actually only returns an approximation.

You can also return multiple values from a function, which will be useful in the project, by separating them by a comma.

```
def divide(n,d):
    return n//d, n%d
```

Doctests

It's a good idea to put text explaining what the file and code does by using what we call a doctest.

We usually put some documentation of what a function does by using triple quotation below the function signature.

```
def square(x):
    """Squares a number.
    >>> square(2)
    4
    >>> square(16)
    256
    """
    return x*x
```

Within the doctest, there is an example interactive session that we can trigger using:

```
python3 -m doctest (filename).py
```

If everything is correct, you should see no output in your terminal. If you'd like more details:

```
python3 -m doctest -v (filename).py
```

which will tell you everything that happened, which will show you what it tried and what happened as a result.

Default Values

When you're defining a function, you can assign a default value that is used when nothing is bound to a formal parameter.

For example,

```
def divide(n,d=10)]
```

will assign d to 10 when no value is given to it by the user. You're not assigning `d` to `10` in the body unless the user passes in no argument. This will also be used in the project as well.

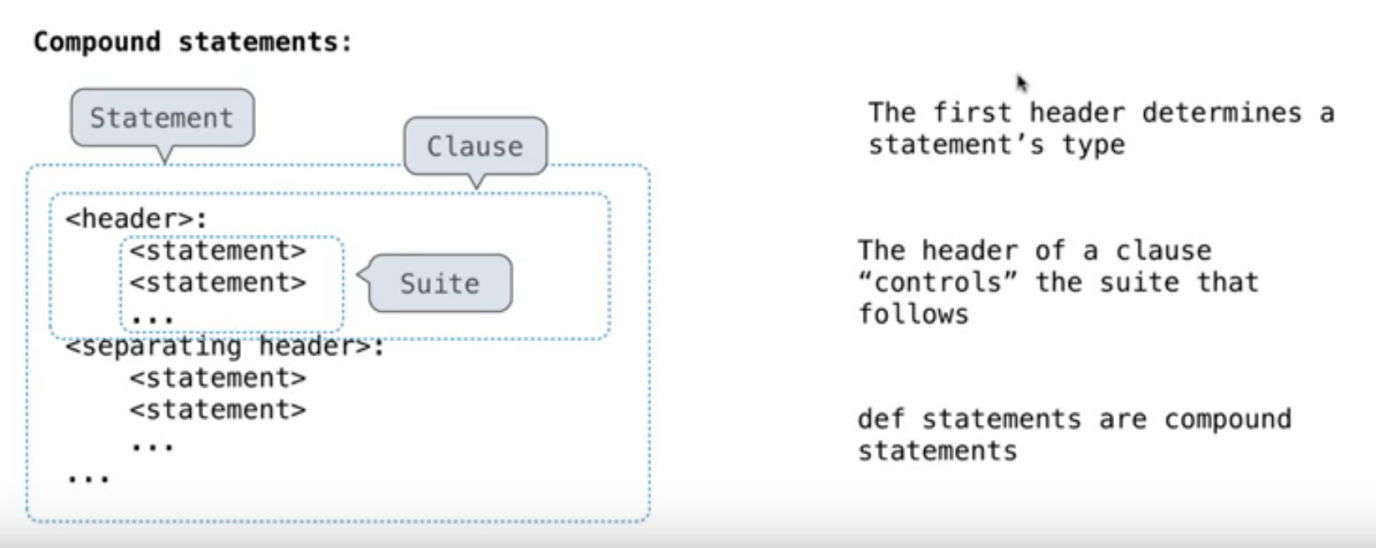
Conditional Statements

First, let's talk about statements. We've seen assignment and `def` statements.

A statement is executed by the interpreter to perform a function.

Statements can be compound:

```
<header>:
    <statement>
    <statement>
<separating header>:
    <statement>
    <statement>
```



The first header determines a statement's type. The header of a caluse "controls" the suite that follows.

`def` statements are compound statements.

A suite is a sequence of statements. To execute a suite means to execute its sequence of statements in order.

Execution rule for a sequence of statements:

1. Execute the first statement.
2. Unless directed otherwise, execute the rest.

The Conditional Statement

For example, let's define the function `abs` from scratch:

```
def abs(x):
    """Return the absolute value of x."""
    if x<0:
```

```
        return -x
    elif x==0:
        return 0
    else:
        return x
```

Taking the execution rule from earlier, each caluse is considered in order. We evaluate the header's expression; if it is a true value, execute the suite and skip the remaining clauses. The remaining clauses are never executed.

> **Syntax Tips**
>
> 1. Always start with an "if" clause.
> 2. You can have zero or more "elif" clauses.
> 3. You can have zero or one "else" clause at the end.

Boolean Contexts

George Boole was a logician and an early founder of computer science. George is interested in Boolean contexts, where all that matters about one particular expression is whether it is true or false.

The earlier code has two Boolean contexts, the if and the elif.

- There are false values in Python: `False` , 0, `''` , `None` , and others that we don't need to know yet.
- Anything else is a true value, so something like `if 3` is the same as saying `if True` .

Iteration

One way we can repeat the same statement many times is a `While` statement. It too is a compound statement, that within its body, contains something we want to execute multiple times.

For example:

```
x=3
While x!=0:
    x=x-1
    print(x)
```

This will count down from `x` to 0.

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the whole suite, then return to step 1.

This also has Boolean contexts like an if statement.

In the example above, notably, we do not stop the suite when `x` becomes 0; we first execute the whole suite, printing 0 along the way, and the checking again if x is 0. It is, so the loop ends when the False value is encountered.

Testing

Earlier, we talked about how we can use doctests to test our programs.

Testing a function is the act of verifying the function's behavior matches expectations. Our language is now sufficiently complex that we need to start testing our implementations.

A **test** is a mechanism for systematically performing this vertification, typically another function containing one or more sample calls to the function being tested. The returned value is then verified against an expected result. Tests involve selecting and validating calls with specific argument values. Tests also serve as documentations, they demonstrate how to call a function and what argument values are appropriate.

Assertions

Programmers use `assert` statements to verify expectations, such as the output of a function being tested. An `assert` statement has an expression in a Boolean context, followed by a quoted line of text in quotes that will be displayed in the expression evaluates to a false value.

```
>>>assert fib(8)==13, 'The 8th Fibonacci number should be 13'
```

When the expressiong being asserted to a true value, executing an `assert` statement has no effect. When it is a false value, `assert` causes an error that halts execution.

A test function for `fib` should have several arguments, including extreme values of `n`.

```
>>> def fib_test():
        assert fib(2)==1, "The 2nd Fibonacci number should be 1"
        assert fib(3)==1, "The 3rd Fibonacci number should be 1"
        assert fib(50)==7778742049, "Error at the 50th Fibonacci number"
```

Tests are usually written either in the same file, or a neighboring file with the suffix `_test.py`, rather than directly into the interpreter.

Doctests can be used to verify interactions with a function.

We can use the doctest module; below, the globals function returns a representation of the global environment, which the interpreter needs in order to evaluate expressions.

```
>>> from doctest import testmod
>>> testmod()
TestResults(failed=0, attempted=2)
```

To verify the doctest interactions for a single function, we use a doctest function called `run_docstring_examples`, which is unfortunately a bit complicated to call. The first argument is the function to test, the second the result of the expression `globals()`, a built-in function that returns the global environment, and the third argument is `True` to indicate we would like "verbose" output: a catalog of all tests run.

When the return value of a function does not match the expected result, the `run_docstring_examples` function will report this problem as a test failure.

When writing Python in files, all doctests in a file can be run with the doctest command line option as discussed earlier.

The key to effective testing is to write and run tests immediately after implementing new functions. It is even good practice to write some tests before you implement, in order to have some example inputs and outputs in your mind. A test that applies a single function is called a **unit test**. Exhaustive unit testing is a hallmark of good program design.