EXPRESSIONS, STATEMENTS, AND FUNCTIONS 1

COMPUTER SCIENCE 61A

September 4, 2012

0.1 Warmup — What Would Python Do?

```
>>> x = 6
>>> def square(x):
...     return x * x
>>> square(x)
```

```
>>> max(pow(2, 3), square(-5)) - square(4)
```

1 Expressions

Expressions describe a computation and evaluate to a value.

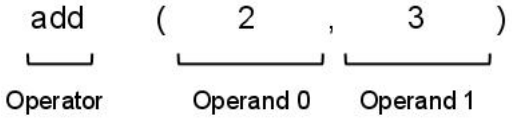
1.1 Primitive Expressions

A *primitive expression* is a single evaluation step: you either look up the value of a name, or take the literal value. For example, numbers, variable names, and strings are all primitive expressions.

```
>>> 2
2
>>> 'Hello World!'
'Hello World!'
```

1.2 Call Expressions

Call expressions are expressions that involve a call to some function. Call expressions are just another type of expression, called a *compound expression*. A call expression invokes a function, which may or may not accept arguments, and returns the function's return value. Recall the syntax of a function call:



Every call expression is required to have a set of parentheses delimiting its comma-separated operands. To evaluate a function call:

1. First evaluate the operator, and then the operands (from left to right).
2. Apply the function (the value of the operator) to the arguments (the values of the operands).

If the operands are nested function calls, apply the two steps recursively.

1.3 Questions

1. Determine the result of evaluating $f(4)$ in the Python interpreter if the following functions are defined:

```
from operator import add

def double(x):
    return x + x

def square(y):
    return y * y

def f(z):
    add(square(double(z)), 1)
```

2. What is the result of evaluating the following code?

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


def so_slow(num):
    x = num
    while x > 0:
        x = x + 1
    return num / 0

square(so_slow(5))
```

3. What will python print?

```
def f():
    print('f')
    return print

def a():
    print('a')
    return "hello"

def b():
    print('b')
    return "world"

f()(a(), b())
```

2 Statements

A statement in Python is executed by the interpreter to achieve an effect.

For example, we can talk about an assignment statement. In an assignment statement, we ask Python to assign a certain value to a variable name. Assignment is a one-line statement, but there are also compound statements! A good example is the `if` statement:

```
if <test>:
    <true_suite>
elif <other_test>:      # shorthand for 'else if'
    <elif_suite>
else:
    <else_suite>
```

Let us consider how Python evaluates an `if` statement — it is different from the evaluation of an expression:

1. Evaluate the header's expression.
2. If it is a true value, execute the suite immediately under the header and then exit the entire `if/elif/else` block.
3. Otherwise, go to the next header (either an `elif` or an `else`). If there is no other header, you have finished.
4. If the header is an `elif`, go back to step 1. Otherwise, if it is an `else`, go to step 2, as if the test was `True`.

Note: Each clause is considered in order.

What is a *suite*? A suite is a sequence of statements or expressions associated with a header. The suite will always have one more level of indentation than its header. To evaluate a suite, we execute its sequence of statements or expressions in order.

We can generalize and say that compound statements comprise headers and suites:

```
<header>:
    <statement>
    <statement>
    ...
<separating-header>:
    <statement>
    ...
```

3 Pure and Non-Pure Functions

Pure function — It only produces a return value (no side effects), and always evaluates to the same result, given the same argument value(s).

Non-Pure function — It produces side effects, such as printing to the screen.

Further in the semester, we will further expand on the notion of a pure function versus a non-pure function.

3.1 One Moar Question

1. What do you think Python will print for the following? `om` and `nom` are defined as follows:

```
>>> def om(foo):
...     return -foo

>>> def nom(foo):
...     print(foo)

>>> nom(4)

>>> om(-4)

>>> save1 = nom(4)
>>> save1 + 1

>>> save2 = om(-4)
>>> save2 + 1
```

4 An Environment Diagram

1. Draw the environment diagram that results from running the following code.

```
n = 7
def f(x):
  n = 8
  return x+1
def g(x):
  n = 9
  return x + 3
def square_result(f, x):
  return f(x) * f(x + 2)
m = square_result(g, n)
```

5 Secrets to Success in CS61A

CS61A is definitely a challenge, but we all want you to learn and succeed, so here are a few tips that might help:

- Ask questions. When you encounter something you dont know, *ask*. That is what we are here for. This is not to say you should raise your hand impulsively; some usage of the brain first is preferred. You are going to see a lot of challenging stuff in this class, and you can always come to us for help.
- Go to office hours. Office hours give you time with the instructor or TAs by themselves, and you will be able to get some (nearly) one-on-one instruction to clear up confusion. You are *not* intruding; the instructors and TAs *like* to teach! Remember that, if you cannot make office hours, you can always make separate appointments with us!
- Do the readings (on time!). There is a reason why they are assigned. And it is not because we are evil; that is only partially true.
- Do (or at least attempt seriously) all the homework. We do not give many homework problems, but those we do give are challenging, time-consuming, and rewarding. The fact that homework is graded on effort does not imply that you should ignore it: it will be one of your primary sources of preparation and understanding.
- Do all the lab exercises. Most of them are simple and take no more than an hour or two. This is a great time to get acquainted with new material. If you do not finish, work on it at home, and come to office hours if you need more guidance!
- Study in groups. Again, this class is not trivial; you might feel overwhelmed going at it alone. Work with someone, either on homework, on lab, or for midterms, as long as you don't violate the cheating policy!
- Most importantly, *have fun!*