EXPRESSIONS, STATEMENTS, AND FUNCTIONS 1

COMPUTER SCIENCE 61A

June 24, 2014

0.1 Warmup — What Would Python Do?

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

Solution: 36

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

Solution: 9

1 Expressions

Expressions describe a computation and evaluate to a *value*. There are two types of expressions, *primitive* and *compound*.

1.1 Primitive Expressions

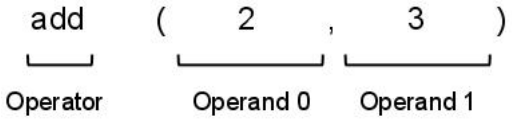
A *primitive expression* is a single evaluation step: you either look up the value of a name, or take the literal value. For example, numbers, variable names, and strings are all primitive expressions.

```
>>> 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 to the operands.

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)
```

Solution: $f(4)$ returns `None`, because f has no return statement.

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

```
from operator import add

def square(x):
    return x * x


def so_slow(num):
    return num
    num / 0

square(so_slow(5))
```

Solution: The function returns 25, the num/0 never executes

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())
```

Solution:

```
f
a
b
hello world
```

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. There are also compound statements but we will cover them later!

We can assign values to names using the = operator

Every assignment will assign the value on the right of the = operator to the name on the left.
For example:

```
>>> foo = 5
```

Binds the value 5 to the name foo.

We can also bind values to names using the import statement.

```
>>> from operator import add
```

Will bind the name add to the add function.

Function definitions also bind the name of the function to that function.

```
>>> def foo():
...     return 0
```

Will bind foo the function defined above

Assignments are also not permanent, if two values are bound to the same name only the assignment executed last will remain.

```
>>> add = 5
>>> from operator import add
```

After the above two statements are executed the value 5 will no longer be bound to add but rather the add function from operator will.

2.1 Questions

1. Determine the result of evaluating $foo(5)$ in the Python interpreter if the following functions are defined in the order below:

```
>>> def bar(param):
...     return param
>>> bar = 6
>>> def foo(bar):
...     return bar(bar)
```

Solution: $foo(5)$ errors because you are trying to apply call expression to the number 5 which is what bar is bound to.

2. Determine the result of evaluating $foo(5)$ in the Python interpreter if the following functions are defined in the order below:

```
>>> def bar(param):
...     return param
>>> zoo = bar
>>> bar = 6
>>> def foo(bar):
...     return zoo(bar)
```

Solution: 5

3. What would python print?

```
>>> from operator import add
>>> def sub(a, b):
...     sub = add
...     return a - b
>>> add = sub
>>> sub = min
>>> print(add(2, sub(2, 3)))
```

Solution: 0

3 Control

Control refers to directing the computer to selectively choose which lines of code get executed. This can mean skipping a portion of code (conditionals) or repeating a portion of code multiple times (iteration).

3.1 Conditional Statements

Conditional statements allow programs to execute different lines of code depending on the current state. A typical if-else set of statements will have the following structure:

```
if <conditional expression>:
    <suite of statements>
elif <conditional clause>:
    <suite of statements>
else:
    <suite of statements>
```

The else and elif statements are optional and you can have any number of elif statements. Here a conditional clause is something that evaluates to either `True` or `False`. The body of statements that get executed are the ones under the first true conditional clause. After a true conditional clause is found, the rest are skipped. Note that in python everything evaluates to `True` except `False`, `0`, `""`, and `None`. There are other things that evaluate to `False` but we haven't learned them yet.

```
>>> if 2 + 3:
...     print(6)
6
```

Here's some example code:

```
>>> def mystery(x):
...     if x > 0:
...         print(x)
...     else:
...        x(mystery)
...
>>> mystery(5)
5
>>> mystery(-1)
TypeError: 'int' object is not callable
```

1. Write a simple function that takes in one input x, whose value is guaranteed to be between 0 and 100. if $x \geq 75$ then print "Q1". If $50 \leq x < 75$ then print "Q2". If $25 \leq xs < 50$ then print "Q3". If $x < 25$ then print "Q4".

```
def find_quartile(x):
```

Solution:

```
if x >= 75:
    print("Q1")
elif x >= 50:
    print("Q2")
elif x >= 25:
    print("Q3")
else:
    print("Q4")
```

2. Now try rewriting the function so that at most 4 lines of code inside the function will ever get executed.

```
def find_quartile(x):
```

Solution:

```
if x >= 50:
    if x >= 75:
        print("Q1")
    else:
        print("Q2")
elif x >= 25:
    print("Q3")
else:
    print("Q4")
```

3.2 Iteration

Using conditional statements we can ignore statements. On the other hand using iteration we can repeat statements multiple times. A common iterative block of code is the while statement. The structure is as follows:

```
while <conditional clause>:
    <body of statements>
```

This block of code literally means while the conditional clause evaluates to $True$, execute the body of statements over and over.

```
>>> def countdown(x):
...     while x > 0:
...         print(x)
...         x = x - 1
...     print("Blastoff!")
...
>>> countdown(3)
3
2
1
Blastoff!
```

1. Fill in the is_prime function to return $True$ if n is a prime and $False$ otherwise. Hint: use the $\%$ operator. $x\%y$ returns the remainder when x is divided by y.

   ```
   def is_prime(n):
   ```

   Solution:

   ```
   k = 2
   while k < n:
       if n % k == 0:
           return False
       k += 1
   return True
   ```

4 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.

4.1 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)
```

Solution: 4

```
>>> om(-4)
```

Solution: 4

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

Solution: `TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'` (We cannot add an integer to `None`.)

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

Solution: 5

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!*