Python – Predict the interpreter's response

```
>>> x = 6
>>> def square(x):
…        return x * x
>>> square()
    ________
>>> from math import sqrt
>>> sqrt(max(pow(2, 3), square(-5)) - square(4))
    ________
```

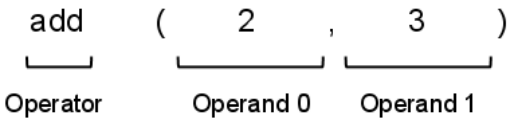
Expressions

Expressions describe a computation and evaluates to a value.

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, function names and strings are all primitive expressions.

```
>>> 2
2
>>> max
<built-in function max>
>>> "Hello World"       #don't worry about Strings yet
'Hello World'
```

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. The point is in the fact that they are calling some function with arguments and returning some value from them. 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. Evaluate the operator and operands – Note that this step makes evaluation a recursive process
2. Apply the function, (the value of the operator), to the arguments, (the values of the operands).

QUESTIONS

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

```
from operators import add
def double(x):
   return x + x

def square(y):
   return y * y

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

2. **Fill in the blanks with the results of evaluating the following code.**

```
>>> def square(x):
...     return x * x
>>> def so_slow(num):
...     so_slow(num + 1)
...     return num / 0
>>> so_slow
    _______________
>>> square(so_slow(-5))
    ___________________
```

Statements

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

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

```
if <test>:
   <true_suite>
elif <other_test>:
   <elif_suite>
else:
   <else_suite>
```

Let's consider how Python evaluates an `if` statement (it is different than 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's 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 means to execute its sequence of statements or expressions in order.

We can generalize and say that *compound statements* are composed of headers and suites like so:

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

Pure Functions vs. Non-Pure Functions (pay attention to domain and range!)

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

Non-Pure function - may produce some side effects (like printing to the screen).

QUESTIONS

3. What do you think Python will print for the following? Assume this definition of the `om` and `nom` procedure:

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

def nom(foo):
    print(foo)

>>> nom(4)
__________
>>> om(-4)

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

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

__________
```

Expression Trees

Now that we have looked at all of these statements, let's go all the way back to expressions! Remember, an expression describes a computation and evaluates to a value. An expression tree helps us break down a call expression into its operator and operand subexpressions in order to illustrate the evaluation process.

QUESTIONS

Draw the expression tree for the following expression calls assuming we have imported the correct functions.

4. `truediv(add(2, mul(3, 4)), 7)`

5. `print(print(abs(-4)))`

First Edition Environment Diagrams

Every expression must be evaluated with respect to an environment, which gives rise to the name "environment model" of evaluation. An environment provides a systematic method for looking up values associated with a name (ie variable). To represent an environment pictorially, we draw environment diagrams.

Environment Diagrams are broken up into two main components:

1. the expression tree
2. environments and values

We have already seen expression trees, but what about "environments and values?" Briefly, we keep an "environment" of bindings from variable to value, and every time we encounter a variable, we look up its value in the current environment. Note that we only lookup the value when we see the variable, and so we'll lookup the value of x only after we've already defined x to be 3.

An **frame** is a box that contains bindings from variables to values. An frame can "extend" another frame; that is, this frame can see all bindings of the frame it extends. We represent this by drawing an arrow from an environment frame to the frame it is extending. The global environment is the only environment that extends nothing.

An **environment** is a series of frames, which we get from extending the current frame. To determine which frames are in an environment, follow the arrows until reaching the global environment. Every frame you passed through is part of that environment. The global environment is only made up of the global frame, because its frame extends nothing.

Here is how to go about using an environment diagram:

1. Start with a box (a frame) labeled with a globe. This box starts out with bindings for all the built-in functions like `+, abs, max, etc` (You don't need to draw these in).
2. Set the current frame to be the global environment. The "current frame" is the first frame of the environment the interpreter is currently using to lookup values. This frame (and the environment) will change as different expressions are evaluated. (You will want to make a mental note of which frame this is at all times) The current environment always
3. Evaluate your expressions, one by one, modifying the environment appropriately.

Some rules to keep in mind when evaluating the following types of expressions:

1. constants: (numbers, strings, etc), they are self-evaluating so don't do any work.
2. variables: try to find a binding for it in the current frame. If no binding exists, follow the arrow to the next frame in the environment, and try to find the binding there, and so on, until you reach the global environment. If it's still not in the global environment, then you can't find it; it's an error! (Recall that the global environment contains all the bindings for primitives).
3. user-defined functions: create a new function object and add the new function definition to the current frame
4. function calls: check if the procedure is a:
    a. primitive - these work by magic, so just apply the procedure intuitively in your head.
    b. User-defined function - evaluate all the arguments as normal. Then, create a new box. Set this new box as the new current frame, and add all the parameters into the box, and have them point to the argument values you evaluated. Evaluate the body of the procedure in the current frame. Once done, go back to the frame from which you made the procedure call.

QUESTIONS

Fill out the environment diagram, the environments and values as well as the expression tree, for the following code:

4.
```
>>> def a(n):
...     return n + 2
...
>>> def b(m):
...     return a(n)/2
...
>>> def e(n):
...     return a(b(n * 2) + 1)
...
>>> n = 5
>>> e(n)
??
```

5.
```
>>> def bar(x):
...     return x + x
...
>>> def foo(x):
...     return bar(add(bar(x),1))
...
>>> garply = foo
>>> garply(5)
??
```

Higher Order Functions

A procedure which manipulates other procedures as data is called a higher order function(HOF). For instance, a HOF can be a procedure that takes procedures as arguments, returns a procedure as its value, or both. You've already seen a few examples, in the lecture notes, so we won't point them out again; let's work on something else instead. Suppose we'd like to square or double every natural number from 0 to n and print the result as we go:

```
def double_every_number(n):



def square_every_number(n):
```

Note that the only thing different about `square_every_number` and `double_every_number` is just what function we call on `n` when we print it. Wouldn't it be nice to generalize procedures of this form into something more convenient? When we pass in the number, couldn't we specify, also, what we want to do to each number `< n`.

To do that, we can define a higher order procedure called `every`. `every` takes in the procedure you want to apply to each element as an argument, and applies it n natural numbers starting from 1. So to write `square_every_number`, you can simply do:

```
def square-every-number(n):
   every(square, n)
```

Equivalently, to write `double-every-number`, you can write

```
def double-every-number(n):
   every(double, n)
```

QUESTIONS

5. Now implement the function `every` which takes in a function `func` and a number `n`, and applies that function to the first n numbers from 1 and prints the result along the way:

   ```
   def every(func, n):
   ```

6. Similarly, try implementing the function `keep`, which takes in a predicate `pred` and a number `n`, and only prints a number from 1 to n to the screen if it fulfills the predicate:

```
def keep(pred, n):
```

Procedures as return values

The problem is often: write a procedure that, given _____, **returns a function** that _____. The keywords – conveniently boldfaced – is that your procedure is supposed to return a procedure. For now, we can do so by defining an internal function within our function definition and then returning the internal function.

```
def my_wicked_procedure(blah):
   def my_wicked_helper(more_blah):
       ...
   return my_wicked_helper
```

That is the common form for such problems but we will learn another way to do this shortly.

QUESTIONS

7. Write a procedure `and_add_one` that takes a function `f` as an argument (such that `f` is a function of one argument). It **should return a function** that takes one argument, and does the same thing as `f`, except adds one to the result.

```
def and_add_one(f):
```

8. Write a procedure `and_add` that takes a function `f` and a number `n` as arguments. It **should return a function** that takes one argument, and does the same thing as the function argument, except adds `n` to the result.

```
def and_add(f, n):
```

9. Python represents a programming community, and in order for things to run smoothly, there are some standards in order to keep things consistent. The following is the recommended style for coding so that collaboration with other python programmers becomes standard and easy. Write your code at the very end:

```
def identity(x):
   return x

def lazy_accumulate(f, start, n, term):
    """
    Takes the same arguments as accumulate from homework and
    returns a function that takes a second integer m and will
    return the result of accumulating the first n numbers
    starting at 1 using f and combining that with the next m
    integers.

    Arguments:
    f - the function for the first set of numbers.
    start - the value to combine with the first value in the
            sequence.
    n - the stopping point for the first set of numbers.
    term - function to be applied to each number before combining.

    Returns:
    A function (call it h) h(m) where m is the number of
    additional values to combine.

    >>> # This does (1 + 2 + 3 + 4 + 5) + (6 + 7 + 8 + 9 + 10)
    >>> lazy_accumulate(add, 0, 5, identity)(5)
    55
    """
```

Secrets to Success in CS61A

- Ask questions. When you encounter something you don't know, ask. That's what we're here for. (Though this is not to say you should raise your hand impulsively; some usage of the brain first is preferred.) You're 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 professor or TAs by themselves, and you'll be able to get some (nearly) one-on-one instruction to clear up confusion. You're NOT intruding; the professors LIKE to teach, and the TAs…well, the TAs get paid. And some like to teach too! Remember that, if you cannot make office hours, you can always make separate appointments with us!
- Do the readings (on time!). There's a reason why they're assigned. And it's not because we're evil; that's only partially true.
- Do all the homeworks. Their being graded on effort is not a license to screw around. We don't give many homework problems, but those we do give are challenging, time-consuming, but very rewarding as well.
- Do all the labwork. Most of them are simple and take no more than an hour or two after you're used to using emacs and Python. This is a great time to get acquainted with new material. If you don't 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!).