EXPRESSIONS AND FUNCTIONS 1

COMPUTER SCIENCE 61A

June 23, 2015

1 Expressions

An expression describes a computation and evaluates to a value.

1.1 Primitive Expressions

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

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

1.2 Call Expressions

A **call expression** applies a function, which may or may not accept arguments. The call expression evaluates to the function's return value.

The syntax of a function call:

Every call expression requires 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 an operand is a nested call expression, then these two steps are applied to that operand in order to evaluate it.

1.3 Questions

1. What will Python print?

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


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

2. What will Python print?

```
>>> from operator import sub, mul
>>> def print_sub(x, y):
...     print('sub')
...     return sub(x, y)
>>> def print_mul(x, y):
...     print('mul')
...     return mul(x, y)
>>> print_sub(print_mul(4, 504), 1)
```

2 Statements

2.1 Assignment Statements

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

For example, an assignment statement assigns a certain value to a variable name:

```
>>> x = 6
```

Here, Python assigns the value of the expression `6` to the name `x`. Since `6` is a primitive (a number), its value is 6. Therefore, Python creates a binding from the name `x` to 6.

2.2 `def` Statements

The `def` statement defines functions:

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

When a `def` statement is executed, Python creates a binding from the name (e.g. `square`) to a function. The variables in parentheses are the function's **parameters** (in this case, `x` is the only parameter). When the function is called, the body of the function is executed (in this case, `return x * x`).

2.3 Questions

1. Determine the result of evaluating the following functions in the Python interpreter:

```
>>> from operator import add
>>> def double(x):
...     return x + x
>>> def square(y):
...     return y * y
>>> def f(z):
...     add(square(double(z)), 1)
>>> f(4)
```

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

```
>>> from operator import add
>>> def square(x):
...     return x * x
>>> def fun(num):
...     return num
...     num / 0
>>> square(fun(5))
```

3. What will Python print?

```
>>> x = 10
>>> def foo():
...     return x
>>> def bar(x):
...     return x
>>> def foobar(new_value):
...     x = new_value
...     y = x + 1
...     return x
>>> foo()


>>> bar(5)


>>> foobar(20)


>>> x


>>> y
```

3 Pure and Non-Pure Functions

1. Pure functions have no side effects – they only produce a return value. They will always evaluate to the same result, given the same argument value(s).
2. Non-pure functions produce side effects, such as printing to your terminal.

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

3.1 Questions

1. What do you think Python will print for the following?

```
>>> def om(cookie):
...     return cookie
>>> def nom(cookie):
...     print(cookie)
>>> om(4)

>>> nom(4)

>>> michelle = om(4)

>>> michelle + 1

>>> brian = nom(4)

>>> brian + 1
```

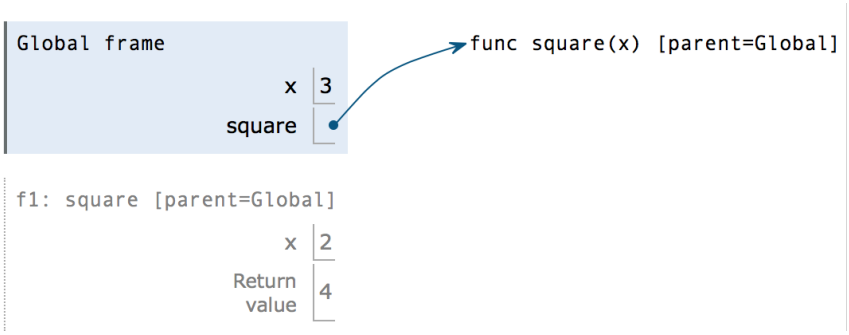
4 Environment Diagrams

An **environment diagram** keeps track of all the variables that have been defined and the values they are bound to.

```
x = 3

def square(x):
    return x ** 2

square(2)
```



When Python executes *assignment statements* (like `x = 3`), it records the variable name and the value:

1. Evaluate the expression on the right side of the = sign
2. Write the variable name and the expression's value in the current frame.

When Python executes *`def` statements*, it records the function name and binds it to a function object:

1. Write the function name (`square`) in the frame and point it to a function object (`func square(x) [parent=Global]`). The `[parent=Global]` denotes the frame in which the function was *defined*.

When Python executes a *call expression* (like `square(2)`), it creates a new frame to keep track of local variables:

1. Draw a new frame. Label it with
   - an index (`f1`)
   - the intrinsic name of the function (`square`)
   - the parent frame (`[parent=Global]`)
2. Bind the formal parameters (e.g. bind `x` to 3).
3. Evaluate the body of the function.

The **intrinsic name** is the name in the function object. For example, if the function object is `func square(x) [parent=Global]`, the intrinsic name is `square`.

If a function does not have a return value, it implicitly returns `None`. Thus, the "Return value" box should contain `None`.

4.1 Questions

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

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

2. Draw the environment diagram so we can visualize exactly how Python evaluates the code.

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