WELCOME TO PYTHON 0

COMPUTER SCIENCE 61A

June 21, 2016

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:

$$\underbrace{\texttt{add}}_{\text{Operator}}\ (\ \underbrace{2}_{\text{Operand 0}}\ ,\ \underbrace{3}_{\text{Operand 1}}\ )$$

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)
```

Solution: 36

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

Solution: 9

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_mul(print_sub(506, 2), 4)
```

Solution:

```
sub
mul
2016
```

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)
```

> **Solution:** `None`
>
> `f(4)` returns `None`, because `f` has no return statement. Note that whenever an expression evaluates to `None`, then the interpreter will not display it.

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))
```

Solution: 25

Note that although `num / 0` would throw an exception *if executed*, this code runs error-free because the function always returns before reaching that line.

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()
```

Solution: 10

Since `x` is not defined in the `foo` environment, then Python looks in the parent frame (the global environment) for the value of `x`.

```
>>> bar(5)
```

Solution: 5

In this case, `x` was passed in as a parameter of `bar`, so since it is defined in the `bar` environment, that value is used.

```
>>> foobar(20)
```

Solution: 20

We define `x` in the `foobar` frame to be `20` and then return that value.

```
>>> x
```

Solution: 10

From the previous call to `foobar`, `x` in the `foobar` frame was defined to be `20`, but the `x` in the global frame was left unchanged. So when we ask for the value of `x` in the global frame, we get `10`.

```
>>> y
```

Solution: `NameError: name 'y' is not defined`

`y` was defined in the `foobar` frame. The global frame does not have `y` defined, nor does it have a parent frame to reference to, so instead Python raises an error.

4. What will Python print?

```
>>> def cake(batter):
...     return batter
>>> def pan(x, y):
...     y = y + 20
...     return x(y)
>>> pan(print, 10)
```

Solution: `30`

We did something tricky here–we passed a function (`print`) into another function (`pan`), where it was bound to the local variable `x`. We then called `x(y)`, which actually executes `print(y)`. Passing functions as arguments is something we'll discuss more in the future.

```
>>> pan(cake, cake(30))
```

Solution: `50`

5. Write some code!

Write a function, `decades_ago`, that takes a year in the past (before 2016) and returns the number of decades that have passed since. A function signature with a *doctest* (an example execution) is below. Fill it in so that the doctest will pass!

```
def decades_ago(year):
    """Returns the number of decades that have passed between
    the year and 2016.

    >>> decades_ago(1995)
    2.1
    """
```

Solution: Many solutions are possible, but here's one:

```
def decades_ago(year):
    years_ago = 2016 - year
    return years_ago / 10
```

3 Side Effects

3.1 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.

3.2 Questions

1. What will Python print for the following?

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

Solution: 4

```
>>> nom(4)
```

Solution: 4

```
>>> michelle = om(-4)
```

Solution: (*nothing*)

Nothing is displayed. This is because the interpreter in an interactive session will print the value of an expression to the terminal by default if the value is not assigned to a variable (unless the value is `None`, in which case nothing is displayed). However, if a value is assigned to the variable, then the value is suppressed.

```
>>> michelle + 1
```

Solution: -3

The previous call to `om(-4)` returns `-4` which is then bound to `michelle`. So, `michelle + 1` is `-3`.

```
>>> brian = nom(4)
```

Solution: 4

Since we're printing the value of `cookie` in `nom`, then the value of `cookie` will always be displayed to the console.

```
>>> brian + 1
```

Solution: `TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'`

Note that although `nom` does not have a return statement, even if we returned the value of `print(cookie)`, the `print` function would return `None` anyways.