Expressions

Primitive expressions, names, and environments

Names

- Giving names to values makes programming easier!
- An assignment statement is one way to bind a name to a value (e.g. x = 1)
- Each name can only be bound to one value
  - Environments keep track of names and their values

Execution Rule for Assignment Statements:
1. Evaluate all expressions to the right of = from left to right.
2. Bind all names to the left of = to those resulting values in the current environment frame.

Primative expressions

- Expressions in programs evaluate to values
- Primitive expressions evaluate directly to values with minimal work needed
  - Numbers (e.g. 42, 3.14, 0)
  - Names (e.g. pi, add)
  - Functions (later today!)
- Some non-primitive expressions: 1 * 2, add(3, 4)

Names

- Giving names to values makes programming easier!
- An assignment statement is one way to bind a name to a value (e.g. x = 1)
- Each name can only be bound to one value
  - Environments keep track of names and their values

Environment diagrams

- Environment diagrams visualize the interpreter’s progress
- Each name is bound to a value
  - A name cannot be repeated in a frame

Announcements

- Set up your computer and all accounts (Lab 0) by today
  - Piazza, Instructional (cs61a-??), OK
- Discussion sections begin today!
- Office hours begin today!
- Homework 0 is due tomorrow (Wednesday) at 11:59pm
- Quiz 1 will be on Thursday at the beginning of lecture
Functions

Call expressions, functions, and `def` statements

---

### Call expressions

- Call expressions use functions to compute a value
- The operator and operands themselves are expressions
- To evaluate this call expression:
  1. Evaluate the operator to get a function value
  2. Evaluate the operands to get their values
  3. Apply the function to the values of the operands to get the final value

```
add (2, 3)
```

### Defining functions

- Functions have inputs and outputs

**Function signature** indicates name and number of arguments

```
def <name>(<parameters>):
    <return expression>
```

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

**Function body** defines computation performed when function is applied

**Execution Rule for `def` Statements:**

1. Create a function with signature `<name>(<parameters>)`:
2. Set the body of that function to be everything indented after the first line
3. Bind `<name>` to that function in the current frame

---

### Calling user-defined functions

**Rules for calling user-defined functions (version 1):**

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```
def square(x):
    return x * x
y = square(-2)
```

**Parameter bound to argument**

---

**Function body defines computation**

**Function signature** indicates name and number of arguments

---

**Calling user-defined functions**

**Rules for calling user-defined functions (version 1):**

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```
def square(x):
    return x * x
y = square(-2)
```

**Intrinsic name of function called**

**Return value (not a binding)!!**
Looking up names in environments

- Every expression is evaluated in the context of an environment
- An environment is a sequence of frames
- So far, there have been two possible environments:
  - The global frame
  - A function's local frame, then the global frame

Rules for looking up names in user-defined functions (version 1):
1. Look it up in the local frame
2. If name isn't in local frame, look it up in the global frame
3. If name isn't in either frame, NameError

Multiple environments (demo)

```python
>>> def square(x):
...     return x * x
>>> y = square(square(-2))
```

None and Print
None means that nothing is returned

- The special value None represents nothing in Python
- A function that does not explicitly return a value will return None
- Note: None is not displayed by the interpreter as the value of an expression

```python
def does_not_square(x):
    ...     x * x

>>> does_not_square(-2)
>>> not_four = does_not_square(-2)
>>> not_four + 4
```

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

None return None means that nothing is returned

- The special value None represents nothing in Python
- A function that does not explicitly return a value will return None
- Note: None is not displayed by the interpreter as the value of an expression

Pure and non-pure functions

- Pure functions just return values
- Non-Pure functions have side effects

```python
def does_not_square(x):
    ...     x * x

>>> does_not_square(-2)
>>> not_four = does_not_square(-2)
>>> not_four + 4
```

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

Nested expressions with print

```python
>>> print(print(1), print(2))
1
None 2
display "1" None, None
```

```text
display "None None"
```

More Functions (demo)

- The operands of a call expression can be any expression
- What about the expression square?

```python
>>> four = describe(square, -2)
Calling function with argument -2
Result was 4
>>> four
4
>>> sixteen = describe(square, four)
Calling function with argument 4
Result was 16
>>> sixteen
16
```