Preconditions

- Precondition is a promise
  - If precondition is true, the function works
  - If precondition is false, no guarantees at all
- Get **software bugs** when
  - Function precondition is not documented properly
  - Function is used in ways that violates precondition

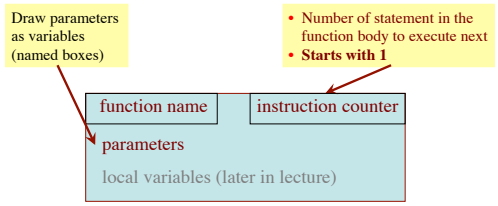
```
>>> to_centigrade(32)
0.0
>>> to_centigrade(212)
100.0
>>> to_centigrade('32')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "temperature.py", line 19 ...
TypeError: unsupported operand type(s)
for -: 'str' and 'int'
```

Precondition violated

How Do Functions Work?

Draw template on a piece of paper

- **Function Frame**: Representation of function call
- A **conceptual model** of Python

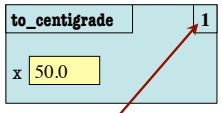


Example: to_centigrade(50.0)

1. Draw a frame for the call
2. Assign the argument value to the parameter (in frame)
3. Execute the function body
   - Look for variables in the frame
   - If not there, look for global variables with that name
4. Erase the frame for the call

```
def to_centigrade(x):
1   return 5*(x-32)/9.0
```

Initial call frame (before exec body)



next line to execute

Call Frames vs. Global Variables

- This does not work:

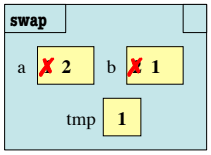
```
def swap(a,b):
    """Swap vars a & b"""
1   tmp = a
2   a = b
3   b = tmp
```

```
>>> a = 1
>>> b = 2
>>> swap(a,b)
```

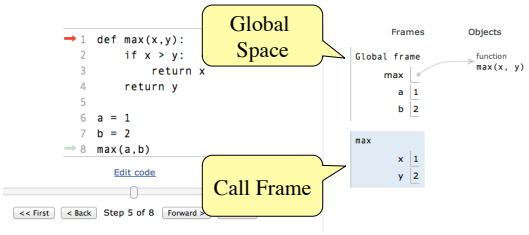
Global Variables

a **1** b **2**

Call Frame



Visualizing Frames: The Python Tutor



Test Cases: Finding Errors

- **Bug**: Error in a program. (Always expect them!)
- **Debugging**: Process of finding bugs and removing them.
- **Testing**: Process of analyzing, running program, looking for bugs.
- **Test case**: A set of input values, together with the expected output.

Get in the habit of writing test cases for a function from the function's specification —even *before* writing the function's body.

```
def number_vowels(w):
    """Returns: number of vowels in word w.

    Precondition: w string w/ at least one letter and only letters"""
    pass  # nothing here yet!
```

Representative Tests

- Cannot test all inputs
  - "Infinite" possibilities
- Limit ourselves to tests that are **representative**
  - Each test is a significantly different input
  - Every possible input is similar to one chosen
- An art, not a science
  - If easy, never have bugs
  - Learn with much practice

Representative Tests for number_vowels(w)

- Word with just one vowel
  - For each possible vowel!
- Word with multiple vowels
  - Of the same vowel
  - Of different vowels
- Word with only vowels
- Word with no vowels

Running Example

- The following function has a bug:

```
def last_name_first(n):
    """Returns: copy of <n> but in the form <last-name>, <first-name>

    Precondition: <n> is in the form <first-name> <last-name>
    with one or more blanks between the two names"""
    end_first = n.find(' ')
    first = n[:end_first]
    last  = n[end_first+1:]
    return last+', '+first
```

- Representative Tests:
  - last_name_first('Walker White')
  - last_name_first('Walker White')

Look at precondition when choosing tests

Unit Test: A Special Kind of Module

- A unit test is a module that tests another module
  - It **imports the other module** (so it can access it)
  - It **imports the cornelltest module** (for testing)
  - It **defines one or more test procedures**
    - Evaluate the function(s) on the test cases
    - Compare the result to the expected value
  - It has special code that **calls the test procedures**
- The test procedures use the cornelltest function

```
def assert_equals(expected,received):
    """Quit program if expected and received differ"""
```

Modules vs. Scripts

Module

- Provides functions, constants
  - **Example**: temperature.py
- import it into Python
  - In interactive shell…
  - or other module
- All code is either
  - In a function definition, or
  - A variable assignment

Script

- Behaves like an application
  - **Example**: helloApp.py
- Run it from command line
  - python helloApp.y
  - No interactive shell
  - import acts "weird"
- Commands *outside* functions
  - Does each one in order

Modules/Scripts in this Course

- Our modules consist of
  - Function definitions
  - "Constants" (global vars)
  - **Optional** application code to call the functions
- All **statements** must
  - be inside of a function or
  - assign a constant or
  - be in the application code
- import should only pull in definitions, not app code

```
# temperature.py
...
# Functions
def to_centigrade(x):
    """Returns: x converted to C"""
...
# Constants
FREEZING_C = 0.0  # temp. water freezes
...
# Application code
if __name__ == '__main__':
    print 'Provide a temp. in Fahrenheit:'
    f = float(raw_input())
    c = round(to_centigrade(f),2)
    print 'The temperature is '+`c`+' C'
```

Testing last_name_first(n)

```
# test procedure
def test_last_name_first():
    """Test procedure for last_name_first(n)"""
    cornelltest.assert_equals('White, Walker',
                              last_name_first('Walker White'))
    cornelltest.assert_equals('White, Walker',
                              last_name_first('Walker  White'))

# Application code
if __name__ == '__main__':
    test_last_name_first()
    print 'Module name is working correctly'
```

Expressions inside of () can be split over several lines.

Quits Python if not equal

Message will print out only if no errors.