eCornell

TOOL

# Fundamental Python Concepts

This tool summarizes key concepts we have covered in this course, including definitions of Python terminology and recommended processes. Use this document as a handy reference to refresh your understanding of functions in Python.

## Executing Functions

### Basic Terminology

**Function call —** using a function in your code

Example:

```
round(26.54)
```

**Arguments —** code on which a function operates

Example:

```
round(26.54)
round(26.54,1)
```

has one argument

has two arguments

**Procedures —** a type of function that can be used as statements but are not expressions. All procedures are functions, but the reverse is not true.

**Fruitful functions —** A type of function that produces a value.

### Print and Return Statements

**Print**

- Displays value on screen
  - Useful for *testing*
  - Not for calculations

Example:

```
def print_plus(n):
    print (n+1)
>>> x = print_plus(2)
3
>>>
```

x [ ]

nothing

**Return**

- Defines function value
  - Needed for *calculations*
  - But does not display

Example:

```
def return_plus(n):
    return (n+1)
>>> x = return_plus(2)
>>>
```

x [ 3 ]

## Local vs. Global Variables

**Local variable:** variable first assigned in the body of a function

**Global variable:** any variable that doesn't appear inside of the body of a function

**Function Definition:**

```
def plus (n):
    ""Returns n+1""
    x=n+1
    return x
```

local var

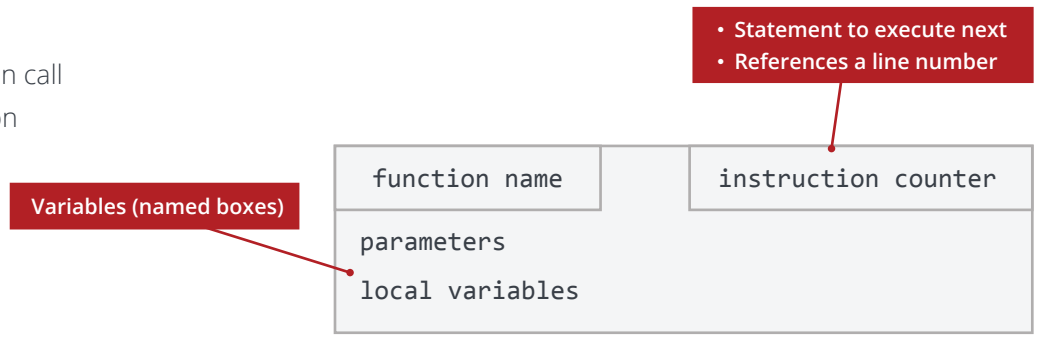
**Function Call:**

```
>>> x = 2
>>> y = plus(4)
```

global var

## Call frame

- A representation of a function call
- A conceptual model of Python




## Call Stack

- Functions are "stacked"
- Cannot remove one above without removing one below
- Affects computer memory, an issue for advanced programs

## Defining Optional Arguments

We can assign default values to arguments

- Write as assignments to parameters in definition
- Parameters with default values are optional

**Example:**

```
p = point_str() # (0,0,0)
p = point_str(1,2,3) # (1,2,3)
```

# Specifying Functions

## Specification

- Purpose is to clearly lay out responsibility
  - What the function promises to do
  - The allowable use of the function
- Requires a formal documentation style

**Example:**

```
def to centigrade(x):
    """Returns:x converted to centigrade

    Value returned has type float.

    Parameter x: temp in fahrenheit
    Precondition: x is a float"""
    return 5*(x-32)/9.0
```

One-line description, followed by blank line

More detail about the function; it may be many paragraphs

Parameter description

Precondition specifies assumptions we make about the arguments

## Precondition

- If true, function must work
- If false, function may or may not work
- Assigns responsibility

Two types of preconditions:

| Type Restrictions | General Preconditions |
|---|---|
| • Ex. **x** is an int<br>• Most common kind<br>  • Guarantees a set of ops<br>  • Some language support<br>• Very easy to check<br>  • **good = type (x) == int** | • Ex. **fname** is a valid file<br>• Less common kind<br>  • Because of function<br>  • Precondition of called functions is second<br>• Not so easy to check |

**Things to Look For**

Are the preconditions clear?

For a fruitful function, is the return result clear?

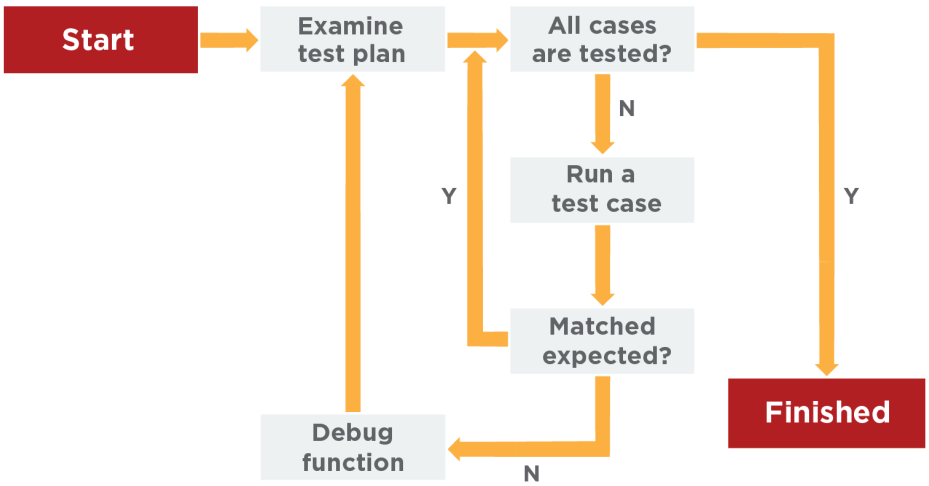
For a procedure, is the outcome clear?

# Testing Functions

## Identifying Errors

- Bug: Error in a program (Always expect them!)
- Debugging: Process of finding & removing bugs
- Testing: Process of analyzing & running a program

## How to test a function




## Selecting test cases

- Choose cases that are representative, or significantly different from each other
- Follow the Rule of Numbers:
  - **Number 1:** The simplest test possible
  - **Number 2:** Add more than was expected
  - **Number 0:** Test something that is missing
- Never test anything that violates the precondition

## Unit test: a script that tests another module

1. Imports the other module
2. Defines one or more test cases
3. Calls the function on each input
4. Compares the result to an expected output

## Types of Testing

| Black Box Testing | White Box Testing |
|---|---|
| • Function is "opaque"<br>  • Test looks at what it does<br>  • Fruitful: what it returns<br>  • Procedure: what changes<br>• Problems:<br>  • Are the tests everything?<br>  • What caused the error? | • Function is "transparent"<br>  • Tests/debugging takes place inside of function<br>  • Focuses on where error is<br>• Problems:<br>  • Much harder to do<br>  • Must remove when done |

# Designing Functions

## Testing First Strategy

- Write the tests first
- Take small steps - make use of placeholders
- Intersperse programming and testing - when you finish a step, test it immediately
- Separate concerns - do not move to a new step until the current one is done

## Function stub

- A function that can be called but is unfinished
- Allows you to test while still working

## Pass

Merely ensures there is a body but says to "do nothing"

Ex. of these two used together:

**Example:**

```
def last_name_first(s):
    """
    Returns: copy of s in form <last-name>, <first-name>
    Precondition: s is in form <first-name> <last-name>
    with one blank between the two names
    """
    pass
```

### Pseudocode

- An outline of steps to carry out in implementing a function definition
- English statements of what you want to do
- Removes conceptual errors

Example:

```
# Find the space between the two names
end_first = introcs.find_str(s,' ' )
# Get the first name
first = s[:end_first]
```

### Stubbed return

- Returns the variable you want to visualize
- Different from the eventual return expression

Example:

```
def last_name_first(s):
    """
    Returns:copy of s in form <last-name>, <first-name>
    Precondition: s is in form <first-name> <last-name>
    with one blank between the two names"""
    end_first = introcs.find_str(s,' ')
    first = s[:end_first]
    # Get the last name
    # Put them together with a comma
    return first     # Not the final answer
```

### Top Down Design

- Function specification is given to you
- Break it up into little problems
- Should not be used too much
- Best used if code is too long or you are repeating yourself a lot

## Enforcing Specifications

### Approaching Error Messages

- Start from the top
- Look at the function call
  - Examine arguments
  - Print if you have to
  - Verify preconditions
- Violation? Error found
  - If not, go to the next call
  - Continue until the bottom

## Assert Statements

Form 1: assert <boolean>

- Does nothing if boolean is True
- Creates an error if boolean is False

Form 2: assert <boolean>, <string>

- Very much like form 1
- But error message includes the message

## repr()

Turns any value into a string

Formatted to represent the original type

**Example:**

```
>>> msg = str(var)+'is invalid'
>>> print(msg)
2 is invalid

>>> msg = repr(var)+ 'is invalid'
>>> print(msg)
'2' is invalid
```

Clear that var is really a string

## Strategy for enforcing preconditions

- Break up preconditions into parts
- Assert the things that are easy to check
- Omit the things that are hard to check

**Example:**

```
def last_name_first(s):
  """Returns:copy of n in form 'last-name, first-name'
  Precondition: n string in form 'first-name last-name'
  There is one or more spaces separating first and last.
  There is no space in either the first or last name"""
  assert type(n) == str  # Check the type
  assert ' ' in n        # Least we can say of space
  # Do not try to enforce anything else
```