§4.1-4.5: Procedures, Functions

21 Sep 2007
CMPT14x
Dr. Sean Ho
Trinity Western University

TRINITY WESTERN UNIVERSITY

Review of last time (§2.6-3.13)

- Formatted output
- abs(), +=, string.capitalize()
- Qualified import

- Selection: if, if..else.., if..elif..else
- Loops: while
  - Sentinel variables
  - Loop counters
  - Using mathematical closed forms instead of loops
- For loops

What's on for today (§4.1-4.3)

- Procedures (functions, subroutines)
  - No parameters
  - With parameters
  - Scope
  - Global variables (why not to use them)
- Functions (return a value)
- Call-by-value vs call-by-reference

Procedures

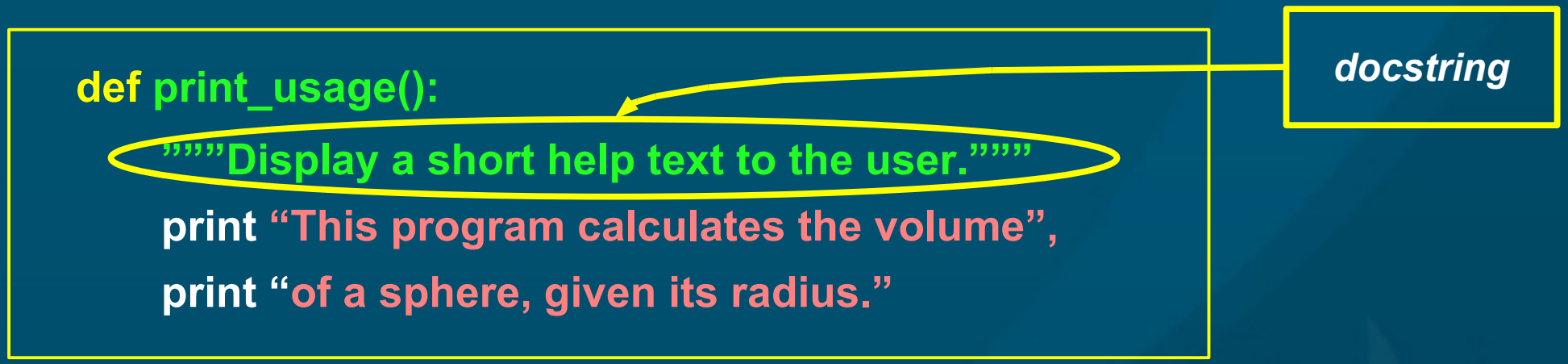
- Fourth program structure/flow abstraction is composition
- This is implemented in Python using procedures
  - Also called functions, subroutines
- A procedure is a chunk of code doing a sub-task
  - Written once, can be used many times
- We've already been using procedures:
  - print, input, raw_input, etc. (not if or while)

Procedure input and output

- Procedures can do the same thing every time:
  - print # prints a new line
- Or they can change behaviour depending on parameters (arguments) input to the procedure:
  - print("Hello!") # prints the string parameter
    - List of parameters goes in parentheses
  - (print is special and doesn't always need parens)
- Procedures can also return a value for use in an expression:
  - numApples = input("How many apples? ")

Example: no parameters

- Procedure to print program usage info:




```
...
if string.capitalize(userInput) == "H":
    print_usage()
```

Example: with parameters

- Calculate volume of a sphere:

```
from math import pi

def print_sphere_volume(radius):
    """Calculate and print the volume of a sphere
    given its radius.
    """
    print "Sphere Volume = %.2f" % (4/3)*pi*(radius**3)

print_sphere_volume(3.5)
```

formal parameter

actual parameter

Scope

- Procedures inherit declarations from enclosing procedures/modules:
  - Declarations:
    - import (e.g., math.pi)
    - variables
    - Other procedures

- Items declared within the procedure are local: not visible outside that procedure
- The scope of a variable is where that variable is visible

Example: scope

```
from math import pi
def print_sphere_volume(radius):
    """Calculate and print the volume of a sphere
    given its radius.
    """
    vol = (4/3)*pi*(radius**3)
    print "Sphere Volume = %.2f" % vol

myRadius = 3.5
print_sphere_volume(myRadius)
```

radius, vol, pi, myRadius

myRadius, pi

- What variables are visible in print_sphere_volume()?
- What variables are visible outside the procedure?

Keep global variables to a minimum

```
from math import pi
def print_sphere_volume(radius):
    """Calculate and print the volume of a sphere
    given its radius.
    """
    myVolume = (4/3)*pi*(radius**3)
    print "Sphere Volume = %.2f" % myVolume

myVolume = 10
print_sphere_volume(3.5)
```

Note assignment to global var

What is the value of myVolume?

Functions

- Functions (function procedures, “fruitful” functions) are procedures which return a value:
  - string.upper('g') returns 'G'
  - ```
    def double_this(x):
        """Multiply by two."""
        return x * 2
    ```
- Statically-typed languages require function definition to declare a return type
- Multiple return statements allowed; first one encountered ends execution of the function
    ```

Functions in Python

- It turns out that in Python, every procedure returns a value
  - **def print_usage():**

    """Print a brief help text."""

    print "This is how to use this program...."
- If no explicit return statement or return without a value, then the special None value is returned
- Must use parentheses when invoking procedures
  - Even those without arguments: **print_usage()**
  - Otherwise you get the function object

Predicates: pre-/post- conditions

```
def ASCII_to_char(code):
    """Convert from a numerical ASCII code
    to the corresponding character.
    """
    return chr(code)
```

- The parameter code needs to be <128: either
  - State preconditions clearly in docstring:
    - """pre: code is an integer between 1 and 128
    - post: returns the corresponding character."""
  - Or code error-checking in the function:
    - if code >= 128:

Example: error-handling

```
def ASCII_to_char(code):
    """Convert from a numerical ASCII code
    to the corresponding character.

    pre: code is an integer
    post: returns the corresponding character
    """
    if (code <= 0) or (code >= 128):
        print "ASCII_to_char(): needs to be <128"
    else:
        return chr(code)
```

Call-by-value and call-by-reference

- In other languages procedures can have side effects: (M2)

```
PROCEDURE DoubleThis(VAR x: INT);
BEGIN
    x := x * 2;
END DoubleThis;

numApples := 5;
DoubleThis(numApples);
```

- Call-by-value means that the value in the actual parameter is copied into the formal parameter
- Call-by-reference means that the formal parameter is a reference to the actual parameter, so it can modify the value of the actual parameter (side effects)

Python is both CBV and CBR

- In M2, parameters are call-by-value
  - Unless the formal parameter is prefixed with "VAR": then it's call-by-reference
- In C, parameters are call-by-value
  - But you can make a parameter be a "pointer"
- Python is a little complicated: roughly speaking,
  - Immutable objects (7, -3.5, False) are call-by-value
  - Mutable objects (lists, user-defined objects) are call-by-reference

Example of CBV in Python

```
def double_this(x):
    """Double whatever is passed as a parameter."""
    x *= 2

numApples = 5
double_this(5)                  # x == 10
double_this(numApples)          # x == 10
double_this("Hello")            # x == "HelloHello"
```

- double_this() has the ability to modify the global numApples, but it doesn't because the changes are only done to the local formal parameter x.

TODO

- Quiz ch2-3 on Mon
- Lab02 due next Wed: 3.14 # 16 / 17 / 23a / 23b / 23c
- Read through §4.13 and Py ch5 for Mon