

Functions!

30 Sep 2010

CMPT140

Dr. Sean Ho

Trinity Western University

Outline for today

- Common **while** loop pitfalls
- **for** loops and **range()**
- **Functions** (procedures, subroutines)
 - **No** parameters
 - With **parameters**
 - **Scope**
 - **Global** variables (and why to avoid them)
 - **Returning** a value
 - **Pre-/post-conditions**

Common errors with loops

- Print **squares** from 1^2 up to 10^2 :
 - ◆ **counter = 0**
 - ◆ **while counter < 10:**
 - **print(counter*counter)**
- What's **wrong** with this loop?
 - **counter** is never **incremented**!
- → Always make sure **progress** is being made in the loop!

Common errors with loops

- Count from 1 up to 10 by twos:
 - ◆ `counter = 1`
 - ◆ `while counter != 10:`
 - `print(counter, end=" ")`
 - `counter += 2`
- What's **wrong** with this loop? How to **fix** it?
 - ◆ `counter = 1`
 - ◆ `while counter < 10:`
 - `print(counter, end=" ")`
 - `counter += 2`

Common errors with loops

- Count from 1.1 up to 2.0 in increments of 0.1:
 - ◆ `counter = 1.1`
 - ◆ `while counter != 2.0:`
 - `print(counter, end=" ")`
 - `counter += 0.1`
- Seems like it should work, but it might not due to inaccuracies in floating-point arithmetic
 - ◆ `counter = 1.1`
 - ◆ `while counter < 2.0:`
 - `print(counter, end=" ")`
 - `counter += 0.1`

for loops

- Many loops do **counting**: the **for** loop is an easy construct that prevents many of these errors
- **Syntax**:
 - ◆ **for target in expression list :**
 - *Statement sequence*
- **Example**:
 - ◆ **for counter in (0, 1, 2, 3, 4):**
 - **print(counter, end=" ")**
 - Output: **0 1 2 3 4**
- for loops can also take an **else** sequence, like while loops

range()

- The built-in function `range()` can generate a list suitable for use in a for loop:
 - `list(range(10))` → `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
 - Note `0-based`, and omits `end` of range
- Specify `starting` value:
 - `list(range(1, 10))` → `[1, 2, 3, 4, 5, 6, 7, 8, 9]`
- Specify `increment`:
 - `list(range(10, 0, -2))` → `[10, 8, 6, 4, 2]`
- *Technically, `range()` produces an `iterator`, which can then be converted to a list. Iterators are beyond the scope of CMPT140.*

for loop examples

- Print **squares** from 1^2 up to 10^2 :
 - **for counter in range(1, 11):**
 - ◆ **print(counter * counter)**
- for loops can iterate over other **lists**:
 - **for appleVariety in ("Fuji", "Braeburn", "Gala"):**
 - ◆ **print("I like", appleVariety, "apples!")**
- *Technically, the for loop uses an **iterator** to get the next item to loop over.*

Program Structure

- Five basic program **structure**/flow abstractions:
 - **Sequence** (newline)
 - **Selection** (if ... elif ... else)
 - **Repetition/loops** (while, for)
 - **Composition** (subroutines)
 - **Parallelism**
- Today we look at **composition**

Functions

- Composition in Python: **functions**
 - Also called procedures, subroutines
- A **function** is a chunk of code doing a **sub-task**
 - **Define** it once
 - **Invoke** (call, use) it many times
- We've already been using functions:
 - **print()**, **input()**, etc. (but **not if** or **while**)
 - e.g., textbook recommends putting all code in a **main()** function

Function input and output

- Functions can do the **same** thing every time:
 - ◆ **print()** **# prints a new line**
- Or can change behaviour depending on input **parameters** (arguments):
 - ◆ **print("Hello!")** **# prints string param**
 - List of parameters goes in **parentheses**
- Functions can also **return** a value for use in an expression:
 - ◆ **name = input("What is your name? ")**
 - This is **not printed** on the screen, but can be **stored** in a variable for later use

Example: no parameters

- Function to print program **usage** info:

```
def print_usage():
```

```
    """Display a short help text to the user."""
```

```
    print("This program calculates the volume"  
          + " of a sphere, given its radius.")
```

docstring

```
...
```

```
userInput = input("Type 'H' for help: ")
```

```
if userInput.upper() == "H":
```

```
    print_usage()
```

Definition

Invocation

- Invocation must use **parentheses** (else you get the function **object**)

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 )
```

*radius,
vol, pi,
myRadius*

```
myRadius = 3.5  
print_sphere_volume(myRadius)
```

*myRadius, pi,
print_sphere_volume()*

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

Avoid global variables

```
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 here?*



Returning a value

- Functions may also **return** a value:
 - **def double_this(x):**
 - ◆ **"""Multiply by two."""**
 - ◆ **return x * 2**
 - e.g., **input()** returns a string
- **Statically**-typed languages require function definition to declare a **return type**
- Multiple **return** statements allowed; first one encountered **ends** execution of the function
- In Python, if **no return**, or return w/o value, then the special **None** value is returned

Example: return value

- Return the volume of a sphere:

```
def sphere_volume(radius):  
    """Find the volume of a  
    sphere given its radius.  
    """  
    from math import pi  
    return (4/3) * pi * (radius**3)
```

*import pi only
for this function*

*not printed,
but returned*

...

```
vol = sphere_volume(3.5)
```

- This is the most **flexible** way of writing this func

Pre- and post-conditions

- Each function's **docstring** should specify:
 - **Pre-condition**: for each input param, the requirements: e.g., type, valid values, ...
 - **Post-condition**: return value and any side-effects produced by the function
- These form a “**contract**” with any code which invokes the function:

```
def sphere_volume(radius):
```

```
    """Find the volume of a sphere given the radius.
```

```
    Pre: radius: positive int or float.
```

```
    Post: returns the volume as a float."""
```