

STRINGS, BRANCHING, ITERATION

VARIABLES (REVISITED)

■ **name**

- descriptive
- meaningful
- helps you re-read code
- cannot be keywords

■ **value**

- information stored
- can be updated

VARIABLE BINDING WITH =

- compute the **right hand side** → **VALUE**
- **store** it (aka **bind** it) in the **left hand side** → **VARIABLE**
- left hand side will be replaced with new value
- = is called assignment

$x = 2$

$x = x * x$

$y = x + 1$

→ Compute value first, then
bind it to variable name; this
will overwrite value of x

BINDING EXAMPLE

- swap variables

– is this ok?

```
x = 1
y = 2
y = x
x = y
```

*This does NOT
do what you
think it does!*

- swap variables

– this is ok!

```
x = 1
y = 2
temp = y
y = x
x = temp
```

TYPES

- variables and expressions
 - `int`
 - `float`
 - `bool`
 - `string` -- NEW
 - ... and others we will see later

STRINGS

- letters, special characters, spaces, digits
- enclose in **quotation marks or single quotes**

```
hi = "hello there"  
greetings = 'hello'
```

- **concatenate** strings

```
name = "eric"  
greet = hi + name  
greeting = hi + " " + name
```

OPERATIONS ON STRINGS

- `'ab' + 'cd'` → **concatenation**
- `3 * 'eric'` → **successive concatenation**
- `len('eric')` → the **length**
- `'eric'[1]` → **indexing**
 - Begins with index 0
 - Attempting to index beyond length - 1 is an error
- `'eric'[1:3]` → **slicing**
 - Extracts sequence starting at first index, and ending before second index
 - If no value before :, start at 0
 - If no value after :, end at length
 - If just :, make a copy of entire sequence

INPUT/OUTPUT: `print`

- used to **output** stuff to console
- keyword is `print`

```
x = 1
print(x)
x_str = str(x)
print("my fav num is", x, ".", "x =", x)
print("my fav num is " + x_str + ". " + "x = " + x_str)
```

INPUT/OUTPUT: `input ("")`

- prints whatever is within the quotes
- user types in something and hits enter
- returns entered sequence
- can bind that value to a variable so can reference

```
text = input("Type anything... ")  
print(5*text)
```

- `input` **returns a string** so must cast if working with numbers

```
num = int(input("Type a number... "))  
print(5*num)
```

IDE's

- painful to just type things into a shell
- better to have a text editor – integrated development environment (IDE)
 - IDLE or Anaconda are examples
- comes with
 - Text editor – use to enter, edit and save your programs
 - Shell – place in which to interact with and run your programs; standard methods to evaluate your programs from the editor or from stored files
 - Integrated debugger (we'll use later)

```
Editor - /Users/ericgrimson/Dropbox (MIT)/Lecture2016New/Lecture2/printExample.py
retirement.py printExample.py getStats.py

1 # -*- coding: utf-8 -*-
2 """
3 Created on Wed Jun  8 11:14:34 2016
4
5 @author: ericgrimson
6 """
7
8
9 x = 1
10 print(x)
11 x_str = str(x)
12 print("my fav num is", x, ".", "x =", x)
13 print("my fav num is " + x_str + "." + "x = " + x_str)
14
```

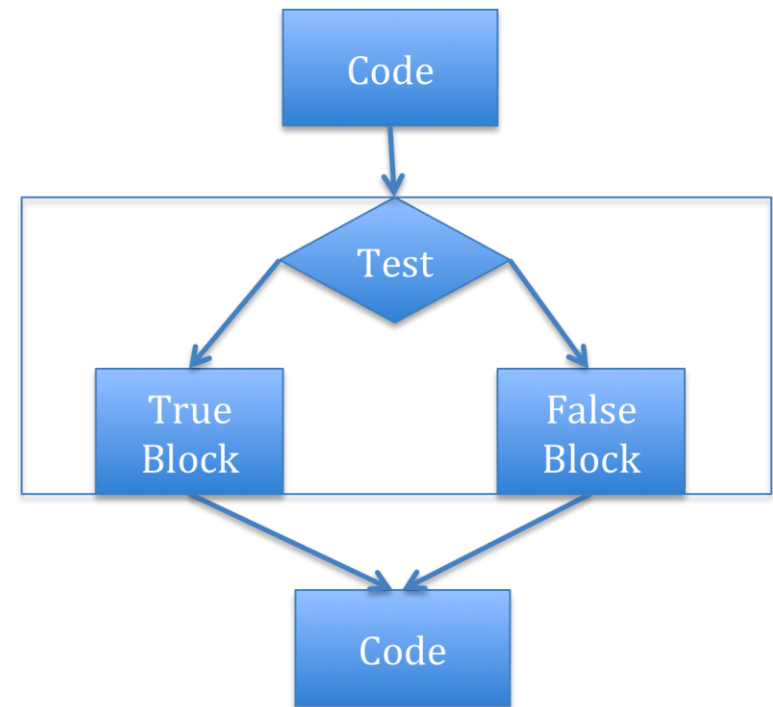
```
IPython console
12: Console 1/A

In [205]: runfile('/Users/ericgrimson/Dropbox
(MIT)/Lecture2016New/Lecture2/printExample.py',
wdir='/Users/ericgrimson/Dropbox (MIT)/Lecture2016New/Lecture2')
1
my fav num is 1 . x = 1
my fav num is 1. x = 1

In [206]:
```

BRANCHING PROGRAMS (REVISITED)

- The simplest branching statement is a **conditional**
 - A test (expression that evaluates to `True` or `False`)
 - A block of code to execute if the test is `True`
 - An optional block of code to execute if the test is `False`



COMPARISON OPERATORS ON `int` and `float`

- `i` and `j` are any variable names

`i > j`

`i >= j`

`i < j`

`i <= j`

`i == j` → **equality** test, True if `i` equals `j`

`i != j` → **inequality** test, True if `i` not equal to `j`

LOGIC OPERATORS ON bools

- a and b are any variable names

`not a` \rightarrow `True` if a is `False`
 `False` if a is `True`

`a and b` \rightarrow `True` if both are `True`

`a or b` \rightarrow `True` if either or both are `True`

CONTROL FLOW - BRANCHING

```
if <condition>:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
elif <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

- <condition> has a value True or False
- evaluate expressions in that block if <condition> is True

USING CONTROL IN LOOPS

- simple branching programs just make choices, but path through code is still linear
- sometimes want to reuse parts of the code
indeterminate number of times

You are in the Lost Forest.



Go left or right?

- You are playing a video game, and are lost in some woods
- If you keep going right, takes you back to this same screen, stuck in a loop

```
if <exit right>:
```

```
    <set background to woods_background>
```

```
    if <exit right>:
```

```
        <set background to woods_background>
```

```
        if <exit right>:
```

```
            <set background to woods_background>
```

```
            and so on and on and on...
```

```
        else:
```

```
            <set background to exit_background>
```

```
    else:
```

```
        <set background to exit_background>
```

```
else:
```

```
    <set background to exit_background>
```

You are in the Lost Forest.



Go left or right?

- You are playing a video game, and are lost in some woods
- If you keep going right, takes you back to this same screen, stuck in a loop

```
while <exit right>:
```

```
    <set background to woods_background>
```

```
<set background to exit_background>
```

CONTROL FLOW:

while LOOPS

```
while <condition>:  
    <expression>  
    <expression>  
    ...
```

- <condition> evaluates to a Boolean
- if <condition> is **True**, do all the steps inside the while code block
- check <condition> again
- repeat until <condition> is **False**

while LOOP EXAMPLE

```
You are in the Lost Forest.  
*****  
*****  
  😊  
*****  
*****  
Go left or right?
```

```
n = input("You are in the Lost Forest. Go left or right? ")  
while n == "right":  
    n = input("You are in the Lost Forest. Go left or right? ")  
print("You got out of the Lost Forest!")
```


CONTROL FLOW:

while and for LOOPS

```
# more complicated with while loop
n = 0
while n < 5:
    print(n)
    n = n+1
```



```
# shortcut with for loop
for n in range(5):
    print(n)
```

*range(5) gives us the integers
0, 1, 2, 3, 4 in turn*

CONTROL FLOW: `for` LOOPS

```
for <variable> in range(<some_num>):  
    <expression>  
    <expression>  
    ...
```

- each time through the loop, `<variable>` takes a value
- first time, `<variable>` starts at the smallest value
- next time, `<variable>` gets the prev value + 1
- etc.

range (start, stop, step)

- default values are `start = 0` and `step = 1` and is optional
- loop until value is `stop - 1`

```
mysum = 0
for i in range(7, 10):
    mysum += i
print(mysum)
```



```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
print(mysum)
```



break STATEMENT

- immediately exits whatever loop it is in
- skips remaining expressions in code block
- exits only innermost loop

```
while <condition_1>:  
    while <condition_2>:  
        <expression_a>  
        break  
        <expression_b>  
    <expression_c>
```

break STATEMENT

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
    if mysum == 5:
        break
print(mysum)
```

- what happens in this program?

for VS while LOOPS

for loops

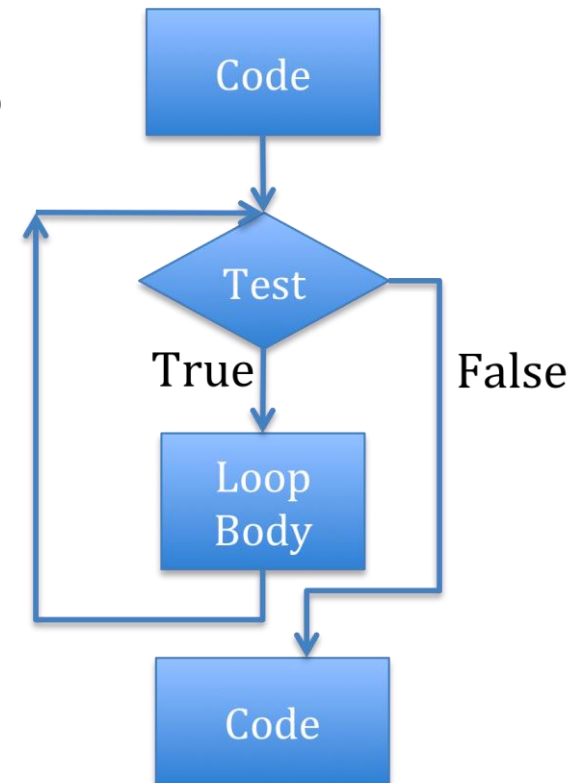
- **know** number of iterations
- can **end early** via `break`
- uses a **counter**
- **can rewrite** a `for` loop using a `while` loop

while loops

- **unbounded** number of iterations
- can **end early** via `break`
- can use a **counter but must initialize** before loop and increment it inside loop
- **may not be able to rewrite** a `while` loop using a `for` loop

ITERATION

- Concept of iteration let's us extend simple branching algorithms to be able to write programs of arbitrary complexity
 - Start with a test
 - If evaluates to `True`, then execute loop body once, and go back to reevaluate the test
 - Repeat until test evaluates to `False`, after which code following iteration statement is executed



AN EXAMPLE

```
x = 3
ans = 0
itersLeft = x
while (itersLeft != 0):
    ans = ans + x
    itersLeft = itersLeft - 1
print(str(x) + '*' + str(x) + ' = ' + str(ans))
```

This code squares the value of x by repetitive addition.

STEPPING THROUGH CODE

```
x = 3
```

```
ans = 0
```

```
itersLeft = x
```

```
while (itersLeft != 0):
```

```
    ans = ans + x
```

```
    itersLeft = itersLeft - 1
```

```
print(str(x) + '*' + str(x) + ' = ' + str(ans))
```

x	ans	itersLeft
3	0	3
	3	2
	6	1
	9	0

Some properties of iteration loops:

- need to set an iteration variable outside the loop
- need to test variable to determine when done
- need to change variable within the loop, in addition to other work

ITERATIVE CODE

- Branching structures (conditionals) let us jump to different pieces of code based on a test
 - Programs are **constant time**
- Looping structures (e.g., while) let us repeat pieces of code until a condition is satisfied
 - Programs now take time that depends on values of variables, as well as length of program

CLASSES OF ALGORITHMS

- Iterative algorithms allow us to do more complex things than simple arithmetic
- We can repeat a sequence of steps multiple times based on some decision; leads to new classes of algorithms
- One useful example are “guess and check” methods

GUESS AND CHECK

- Remember our “declarative” definition of square root of x
- If we could guess possible values for square root (call it g), then can use definition to check if $g * g = x$
- We just need a good way to generate guesses

FINDING CUBE ROOT OF INTEGER

- One way to use this idea of generating guesses in order to find a cube root of x is to first try 0^{**3} , then 1^{**3} , then 2^{**3} , and so on
- Can stop when reach k such that $k^{**3} > x$
- Only a finite number of cases to try

SOME CODE

```
x = int(input('Enter an integer: '))
ans = 0
while ans**3 < x:
    ans = ans + 1
if ans**3 != x:
    print(str(x) + ' is not a perfect cube')
else:
    print('Cube root of ' + str(x) + ' is ' + str(ans))
```


EXTENDING SCOPE

- Only works for positive integers
- Easy to fix by keeping track of sign, looking for solution to positive case

SOME CODE

```
x = int(input('Enter an integer: '))
ans = 0
while ans**3 < abs(x):
    ans = ans + 1
if ans**3 != abs(x):
    print(str(x) + ' is not a perfect cube')
else:
    if x < 0:
        ans = - ans
    print('Cube root of ' + str(x) + ' is ' + str(ans))
```

LOOP CHARACTERISTICS

- Need a loop variable
 - Initialized outside loop
 - Changes within loop
 - Test for termination depends on variable
- Useful to think about a **decrementing function**
 - Maps set of program variables into an integer
 - When loop is entered, value is non-negative
 - When value is ≤ 0 , loop terminates, and
 - Value is decreased every time through loop
- Here we can use `abs(x) - ans**3`

WHAT IF MISS A CONDITION?

- Suppose we don't initialize the variable?
 - Likely get a `NameError`; or worse use an expected value to initiate the computation
- Suppose we don't change the variable inside the loop?
 - Will end up in an infinite loop, never reaching the terminating condition

GUESS-AND-CHECK

- you are able to **guess a value** for solution
- you are able to **check if the solution is correct**
- keep guessing until find solution or guessed all values
- the process is **exhaustive enumeration**

CLEANER GUESS-AND-CHECK

– cube root

```
cube = 8
```

```
for guess in range(cube+1):
```

```
    if guess**3 == cube:
```

```
        print("Cube root of ", cube, " is ", guess)
```

CLEANER GUESS-AND-CHECK – cube root

```
cube = 8

for guess in range(abs(cube)+1):
    if guess**3 >= abs(cube):
        break
if guess**3 != abs(cube):
    print(cube, 'is not a perfect cube')
else:
    if cube < 0:
        guess = -guess
    print('Cube root of ' + str(cube) + ' is ' + str(guess))
```

EXHAUSTIVE ENUMERATION

- Guess and check methods can work on problems with a finite number of possibilities
- Exhaustive enumeration is a good way to generate guesses in an organized manner