CMPT 120
Control Structures in Python

Summer 2012

Instructor: Hassan Khosravi

The If statement

- The most common way to make decisions in Python is by using the if statement.
- The if statement allows you ask if some condition is true. If it is, the body of the if will be executed.
- Boolean Expressions:
  - The expressions that are used for if conditions must be either true or false.
  - The indented print statements are not executed when the if condition is false.
- answer = int(raw_input("input number 2 please: "))
- if answer == 2:
  - == and != are used for comparison
  - = is used for assignment
  - = being used for comparison raises error

Example

- Write a program to check whether input is 2 or not

- answer = int(raw_input("input number 2 please: "))
- if answer == 2:
- print "You got it right!"
- if answer != 2:
- print "Sorry, wrong answer!"
- print "That is the end of the game."

Example Odd or Even

- decide whether a number is odd or even.

- number = raw_input("enter a number: ")
- if int(number)/2 == float(number)/2:
- print "Even"
- else:
- print "Odd"

Example Maximum of 3 Numbers

- Read three different integers and determine their maximum.
  - num1 = int(raw_input("enter first number: "))
  - num2 = int(raw_input("enter second number: "))
  - num3 = int(raw_input("enter third number: "))
  - if num1 > num2:
  - if num1 > num3:
  - print num1, "is the biggest"
  - else:
  - print num3, "is the biggest"
  - else:
  - if num2> num3:
  - print num2, "is the biggest"
  - else:
  - print num3, "is the biggest"

The For Statement

- The easiest way to construct a for loop is with the range function.
  - When a for loop is given range(x), the loop body will execute x times.
  - The range starts from zero and counts up to num − 1.
    - num = int(raw_input("How high should I count? "))
    - for i in range(num):
    - print i,
  - To count from one to n, we could have written the loop like this:
    - num = int(raw_input("How high should I count? "))
    - for i in range(num):
    - print i+1,

The For Statement

- write “Enter a nonnegative integer:”
- read n
- set factorial to 1
- do this for i equal to each number from 1 to n:
- set factorial to factorial × i
- write factorial

- n = int(raw_input("Enter a nonnegative integer: "))
- factorial = 1
- for i in range(n):
- `    factorial = factorial * (i+1)`
- print factorial

Average

- 1. set sum to 0
- 2. for i equal to each number from 1 to 10
- 3. read num
- 4. set sum to sum + num
- 5. average = sum/10
- 6. write average

- sum1 = 0
- for i in range(10):
- num = int(raw_input("enter number " + str(i+1) + ": "))
- sum1 = int(sum1) + int(num);
- final = float(sum1)/10
- print final

Prime

- Read a value and determine whether it is prime or not
- write "read number"
- 2. read n
- 3. set prime to True
- 4. for i from 1 to round(n/2)
- 5. If remainder of n/i is 0
- 6. set prime to False
- 7. if prime = True
- 8. write "your number is prime"
- 9. if prime = False
- 10. write "your number is not prime“

- input = int(raw_input("Enter a number to see whether it is prime or not: "))
- result = True
- newinput = int(round(input/2))
- for i in range(newinput):
- if input%(i+2) == 0:
- result = False
- print result

The While statement

- To construct a while loop, you use a condition as you did in a if statement.
- The body of the loop will execute as many times as necessary until the condition becomes false.

- name = raw_input("What is your name? ")
- while name=="":
- name = raw_input("Please enter your name: ")
- print "Hello, " + name

- When you use an indefinite loop, you have to make sure that the loop condition eventually becomes false. If not, your program will just sit there looping forever. This is called an infinite loop.
- x =1
- while x==1:
- print "x is 1“
- Press control-C to stop

Sum of positive integers

- sum of some positive numbers
- 2. write "Enter some numbers"
- 3. set sum to 0
- 4. read number
- 5. while number >0
- 6. sum = sum +num
- 7. read number
- 8. write "sum of numbers are"
- 9. write sum

- sum =0
- num = int(raw_input("enter number: "))
- while num >= 0:
- sum = sum + num
- num = int(raw_input("enter number: "))
- print "the total sum is", sum

Choosing Control Structures

- Just do it.
  - statements in Python are executed in the order that they appear.
- Maybe do it.
  - If you have some code that you want to execute only in a particular situation, then a conditional (if statement) is appropriate
  - if will run its body zero times or one time.
  - If you need to do similar things more than once, you should be looking at a loop.
- Do it this many times.
  - you don’t need to know how many times you’ll loop when you’re writing the program.
  - You just need to be able to figure this out when the program gets to the for loop.

- Do it until it’s done.
  - There are often situations when you can’t tell how many times to loop until you notice that you’re done

Perfect number

- Determine whether a number is a perfect number or not

```
for number in range(1,101):
    sum_divisor=0
    for i in range(number-1):
        if number%(i+1) == 0:
            sum_divisor = sum_divisor + i+1
    if number == sum_divisor:
        print number, "is a perfect number"
```

Total amount of money in pennies.

- pennies = int(raw_input("How many pennies do you have?: "))
- nickels = int(raw_input("How many nickels do you have?: "))
- dimes = int(raw_input("How many dimes do you have?: "))
- quarter = int(raw_input("How many quarters do you have?: "))
- loonies = int(raw_input("How many loonies do you have?: "))
- toonies = int(raw_input("How many toonies do you have?: "))
- sum =pennies + nickels*5 + dimes*10 + quarter*25 + loonies*100 + toonies*200
- print "you have", sum, "pennies"

Total amount of money in pennies. Not using integers

- pennies = raw_input("How many pennies do you have?: ")
- nickels = raw_input("How many nickels do you have?: ")
- dimes = raw_input("How many dimes do you have?: ")
- quarter = raw_input("How many quarters do you have?: ")
- loonies = raw_input("How many loonies do you have?: ")
- toonies = raw_input("How many toonies do you have?: ")
- sum =pennies + nickels*5 + dimes*10 + quarter*25 + loonies*100 + toonies*200
- print "you have", sum, "pennies"

Guessing game

- Determine whether the guess is 50 between 1-49 or 51-100

- print "Think of a number from 1 to 100."
- smallest = 1
- largest = 100
- guess = (smallest + largest) / 2
- answer = raw_input("Is your number 'more', 'less',"
- " or 'equal' to " + str(guess) + "? ")
- if answer == "more":
- smallest = guess + 1
- elif answer == "less":
- largest = guess - 1
- print smallest, largest

Trying the loop

```
print "Think of a number from 1 to 100."
smallest = 1
largest = 100
while answer != "equal":
  guess = (smallest + largest) / 2
  answer = raw_input( "Is your number 'more', 'less'," \
  " or 'equal' to " + str(guess) + "? " )
  if answer == "more":
    smallest = guess + 1
  elif answer == "less":
    largest = guess - 1
  print smallest, largest
print "I got it! "
```

- Initialize answer

comments

- Comments can be used in your code to describe what's happening.
- In Python, the number sign (#, also called the hash or pound sign) is used to start a comment.
  - Everything on a line after the # is ignored:

- The comment should explain what is happening and/or why it needs to be done.

comments

```
print "Think of a number from 1 to 100."
# start with the range 1-100
smallest = 1
largest = 100
# initialize answer to prevent NameError
answer = ""
while answer != "equal":
    # make a guess
    guess = (smallest + largest) / 2
    answer = raw_input( "Is your number 'more', 'less', "   " or 'equal' to " +
str(guess) + "? " )
    # update the range of possible numbers
    if answer == "more":
        smallest = guess + 1
    elif answer == "less":
        largest = guess - 1
print "I got it!"
```

comments

- Often, when beginning programmers are told “comments are good,” the results are something like this:
  - # add one to x
  - x = x + 1
- Anyone reading your code should understand Python, and doesn’t need the language explained to them
- Comments that actually explain how or why are much more useful:
  - # the last entry was garbage data, so ignore it
  - count = count - 1

comments

- It is often useful to put a comment at the start of each control structure, explaining what it does
  - # if the user entered good data, add it in
  - if value >= 0:
  - total = total + value
  - count = count + 1

docstring

- Docstrings is longer comments that may take a few lines. It is nice to add docstrings to the beginning of functions

```
def middle_value(a, b, c):
    """
    Return the median of the three arguments. That is,
    return the value that would be second if they were
    sorted.
    """
    if a <= b <= c or a >= b >= c:
        return b
    elif b <= a <= c or b >= a >= c:
        return a
    else:
        return c

print middle_value(2, 3, 4)
```

docstring

```
def stars(num):
    """
    Return a string containing num stars.
    This could also be done with "*" * num, but that
    doesn't demonstrate local variables.
    >>> print stars(5)
    *****
    >>> print stars(15)
    ***************
    """
    starline = ""
    for i in range(num):
       starline = starline + "*"
    return starline


num = int(raw_input("How many lines should I print? "))
for i in range(num):
  print stars(i+1)
```

Running Time

- In this section, we will explore how long it takes for algorithms to run.
- The running time of an algorithm will be part of what determines how fast a program runs.
- Let's compare two algorithms for the guessing game
  - Method A:The one we have did in class
  - Method B: starts at zero and guesses 1, 2, 3, . . . , until the user finally enters "equal"
- What's different about this algorithm is the amount work it has to do to?

How can we compare code?

- Lines of Code?
- How long it takes the programmer to write the program?
- How long it takes to run?
  - What parameter makes the program harder (longer)

- How many steps the program requires?
  - The bottleneck of the algorithm
    - The expressions that in the inner most loops
      - Check the guessing game code

Number of “Steps”

- Try counting the bottleneck without bothering about small details.

- statement 1
- for i from 1 to log n:
- ⠀⠀⠀⠀statement 2
- ⠀⠀⠀⠀for j from 1 to n/2:
- ⠀⠀⠀⠀⠀⠀⠀⠀statement 3

- Here, “statement 3” is in the innermost loop, so we will count the number of times it executes.
- The first for loop runs log n times, and the second runs n/2 times. So, the running time is (log n) · (n/2).
- We discard the constant factor and get a running time of n log n.

Running

- Suppose we were writing a guessing game that guessed a number from 1 to n
  - What is the minimum number of possible steps?
    - Method A : 1
    - Method B: 1
  - What the maximum number of steps?
    - Method A: ?
    - Method B: n
- Each time the algorithm makes a guess, it chops the range from smallest to largest in half.
- The number of times we can cut the range 1to n in half before getting down to one possibility is $\lceil \log_2 n \rceil$.

logarithm

- The mathematical expression $\log_2 n$ is the “base-2 logarithm of n”.
- It’s the power you have to raise 2 to get n.
  - $x = \log_2 n$, ➔ $2^x = n$.
  - log2 1 = 0 ,
  - log2 16 = 4 ,
  - log2 1024 = 10 ,
  - log2 1048576 = 20 .
- We could give this program inputs with n = 1000000 and it would still only take about 20 steps.

- Why does this algorithm take about $\log_2 n$ steps
- Consider the number of possible values that could still be the value the user is thinking of.
- Remember that this algorithm cuts the number of possibilities in half with each step

Step	Possible values
0	$n = n/2^0$
1	$n/2 = n/2^1$
2	$n/4 = n/2^2$
3	$n/8 = n/2^3$
k	$n/2^k$

- In the worst case, the game will end when there is only one possibility
  - $n/2^k = 1 \rightarrow n = 2^k$
  - $k = \log_2 n$

Running

- What is the average number of steps required?
  - Method A: hard to determine
  - Method B: 50
- In practice the maximum number of steps is usually taken into consideration
  - Hard to find average
  - Reliability

Repeated Letters

- Write an algorithm that checks a word to see whether it has any repeated words or not.
  - Input → “this”
    - Output → “No repetition”
  - Input “that”
    - Output → “There are repetitions”

Repeated Letters

- Method:
  - Take each word and compare it to all the letters on the right side of it
    - For input this
      - t is compared with h,i,s the letter is not repeated
      - h is compared with i,s the letter is not repeated
      - i is compared with s the letter is not repeated
      - s is compared with nothing the letter is not repeated
  - Take each word and compare it to all the letters on the right side of it
    - For input that
      - t is compared with h,a,t the letter is repeated
      - h is compared with a,t the letter is not repeated
      - a is compared with t the letter is not repeated
      - t is compared with nothing the letter is not repeated

Repeated Letters

- write “Enter the word:”
- read word
- set counter to 0
- for all letters letter a in the word, do this:
- for all letters letter b to the right of letter a, do this:
- if letter a is equal to letter b then
- set counter to counter+1
- if counter > 0 then
- write “There are repetitions”
- else
- write “No repetitions”

```
word = raw_input("Enter the word: ")
counter = 0
length = len(word)
for i in range(length): # for each letter in the word...
    for j in range(i+1, length):  # for each letter after that one...
        if word[i]==word[j]:
            counter = counter + 1
if counter>0:
    print "There are repeated letters"
else:
    print "There are no repeated letters"
```

Run time of repeated words

- What is the run time of this program?
  - What is the bottleneck?
- This program makes $n(n-1)/2 = n^2/2 - n/2$ comparisons if you enter a string with n characters.
  - How many statement?
- We would say the runtime is n^2

Subset Sum

- Let's consider one final example where the best known solution is very slow.
  - we get a list of integers from the user and are asked if some of them (a subset) add up to a particular target value.
  - This problem is known as "subset sum",
  - Input:
    - Set of numbers {6, 14, 127, 7, 2, 8}
    - Target = 16
  - Output:
    - Yes → since 6+2+8 = 16
  - Input:
    - Set of numbers {6, 14, 127, 7, 2, 8}
    - Target = 12
  - Output:
    - No

Subset Sum

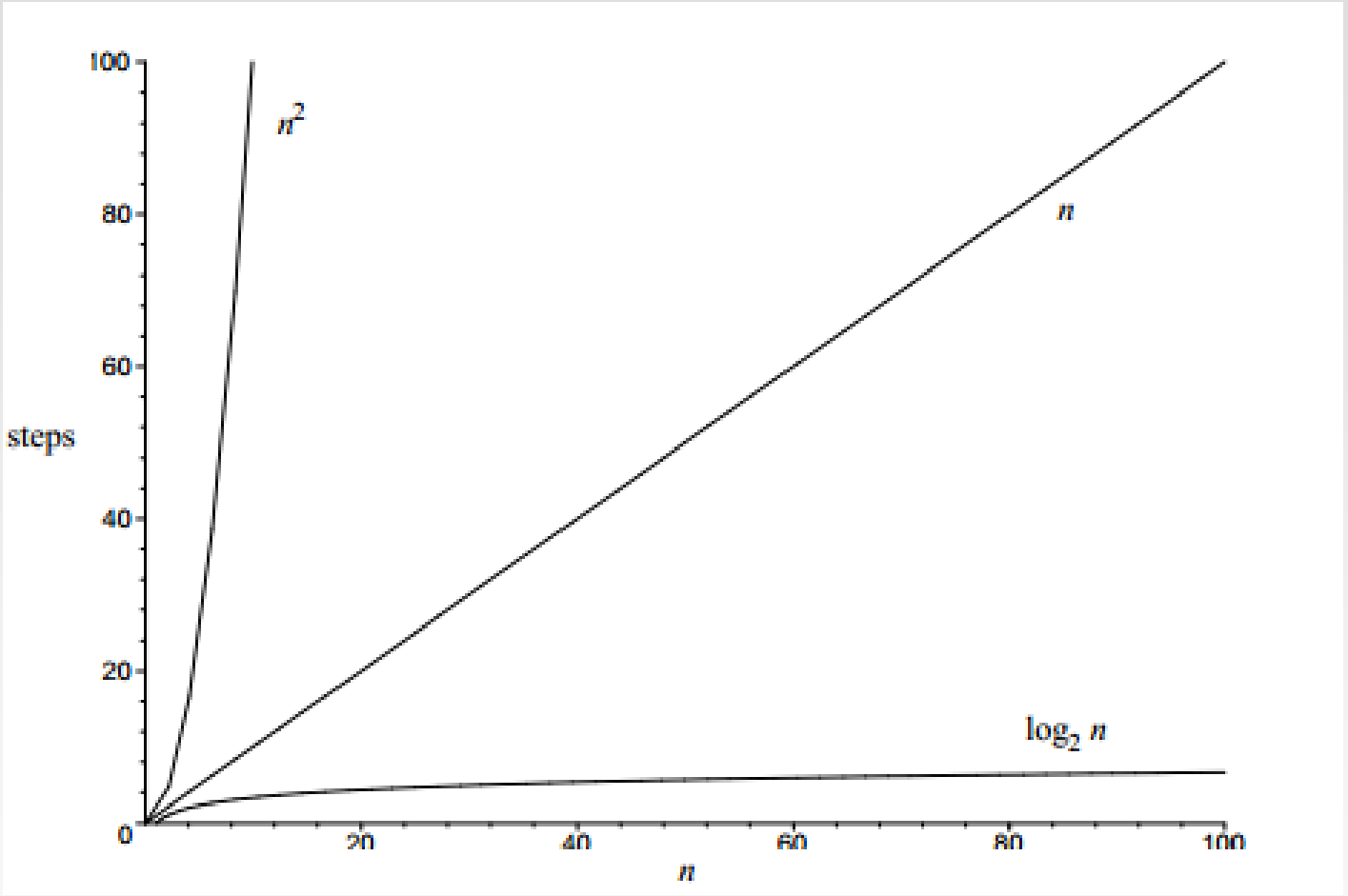
- for every subset in the list:
- set sum to the sum of this subset
- if sum is equal to target:
- answer “yes” and quit
- answer “no”

- What is the running time of this algorithm?

n	$2^n \approx$	Approx. time
4	16	16 milliseconds
10	10^3	1 second
20	10^6	17.7 minutes
30	10^9	11.6 days
40	10^{12}	31.7 years

```
import itertools
def findsubsets(S,m):
    return set(itertools.combinations(S, m))

input = [1,2,3,4]
for i in range(1,len(input) +1):
    print findsubsets(input,i),
```


100
80
60
40
20
0
steps
n^2
n
$\log_2 n$
20
40
60
80
100
n

Debugging

- programming errors are called bugs and the process of tracking them down and correcting them is called debugging.
- Three kind of errors
  - Syntax errors
  - Runtime errors
  - Semantic errors

Syntax Error

- Python can only execute a program if the program is syntactically correct; The process fails and returns an error message if there are any errors.
  - For example, in English, a sentence must begin with a capital letter and end with a period
    - this sentence contains a syntax error.
  - For most readers, a few syntax errors are not a significant problem
    - Python is not so forgiving.
    - If there is a single syntax error anywhere in your program, Python will print an error message and quit

Runtime errors

- The second type of error is a runtime error, so called because the error does not appear until you run the program.
  - Errors that you get based on input during run time

Semantic errors

- If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages
  - But it will not do the right thing
  - Identifying semantic errors can be very tricky
  - When you realize there's a problem with your program, you should do things in this order:
    - 1. Figure out where the problem is.
      - Use print statements to narrow down the possibilities
    - 2. Figure out what's wrong.
      - a variable doesn't contain the value you think it should
      - flow of control isn't the way it should be
    - 3. Fix it

Reading

- Read Topic 3 from Introduction to Computing Science and Programming I

- Read Sections 4.2–4.7, 6.2–6.4, 1.3, How to Think Like a Computer Scientist.