## CSC 221: Computer Programming I

## Fall 2011

### Python control statements

- operator precedence
- importing modules
- random, math
- conditional execution: if, if-else, if-elif-else
- counter-driven repetition: for
- conditional repetition: while



## Recall: Python functions

### recall the general form of a Python function

```
def FUNCTION_NAME(PARAM1, …, PARAM2):
    """doc string that describes the function"""
    STATEMENTS
    return OUTPUT_VALUE                  # optional
```

EXERCISE: define a function that, given the current temperature and wind speed, calculates the wind chill formula

$$\text{wind chill} = 35.74 + 0.6215*\text{temp} + (0.4275*\text{temp} - 35.75)*\text{wind}^{0.16}$$

## Complex expressions

how do you evaluate an expression like `0.4275*temp-35.75 * wind**0.16`

Python has rules that dictate the order in which evaluation takes place

- ** has higher precedence, followed by * and /, then + and –
- meaning that you evaluate the part involving ** first, then * or /, then + or –

```
1 + 2 * 3 → 1 + (2 * 3) → 1 + 6 → 7

2 ** 10 – 1 → (2**10) – 1 → 1024 – 1 → 1023
```

- given operators of the same precedence, you evaluate from left to right

```
8 / 4 / 2 → (8 / 4) / 2 → 2 / 2 → 1
```

GOOD ADVICE: don't rely on these (sometimes tricky) rules

- place parentheses around sub-expressions to force the desired order

```
(0.4275*temp – 35.75)*(wind**0.16)
```

3

## Python modules

we have seen how to use the IDLE editor to create a module/file of functions

- can then load those functions into the interpreter shell via "Run Module"

alternatively, can use the `import` statement to load a module

```
from MODULE import FUNCTION1, FUNCTION2, …
```

e.g.

```
from intro import feetToMeters, metersToFeet

from intro import *
```

(* will match and load all functions in the module)

4

# Built-in modules: random

Python provides many useful modules,

- e.g., the `random` module contains functions for generating random values

`randint(low, high)` returns a random integer in range [low, high]

`random()` returns a random real in range [0, 1)

`choice([option1, …, optionN])` returns a random value from the list [option1, …, optionN]

```
from random import randint, choice

def rollDie(numSides):
    return randint(1, numSides)

def flipCoin():
    return choice(["heads", "tails"])
```

5

# Built-in modules: math

- e.g., the `math` module contains common math functions and constants

`sqrt(num)` returns $\sqrt{\text{num}}$
`ceil(num)` returns $\lceil \text{num} \rceil$
`floor(num)` returns $\lfloor \text{num} \rfloor$
`log(num, base)` returns $\log_{\text{base}} \text{num}$
`pi` the value π = 3.14159…

EXERCISE: define a function for calculating the distance between the points ($x_1$, $y_1$) and ($x_2$, $y_2$)

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
from math import sqrt

def distance(x1, y1, x2, y2):
    ???
```

6

# Conditional execution

so far, all of the statements in methods have executed *unconditionally*
- when a method is called, the statements in the body are executed in sequence
- different parameter values may produce different results, but the steps are the same

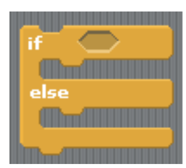
many applications require *conditional execution*
- different parameter values may cause different statements to be executed

for example, consider the `windChill` formula
- the formula only applies when wind speed > 3 mph
- if wind speed is ≤ 3 mph, wind chill is the same as the temperature

$$\text{wind chill} = \begin{cases} \text{temp} & \text{if wind} <= 3 \\ 35.74 + 0.6215 * \text{temp} + (0.4275 * \text{temp} - 35.75) * \text{wind} ** 0.16 & \text{otherwise} \end{cases}$$

7

# If statements




in Python, an *if statement* allows for conditional execution
- i.e., can choose between 2 alternatives to execute

```
if TEST_CONDITION:
    STATEMENTS_TO_EXECUTE_IF_TEST_IS_TRUE
else:
    STATEMENTS_TO_EXECUTE_IF_TEST_IS_FALSE
```

```
def windChill(temp, wind):
    if wind <= 3:
        return temp
    else:
        chill = 35.74 + 0.6215*temp + \
                (0.4275*temp - 35.75)*(wind**0.16)
        return chill
```

if the test is true (wind ≤ 3), then this statement is executed

otherwise (wind > 3), then these statements are executed

note: the \ character is used to break a statement across lines

8

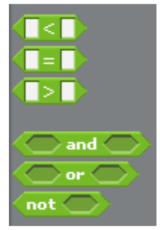
# Boolean operators

## standard relational operators are provided for the if test

| | | | |
|---|---|---|---|
| `<` | less than | `>` | greater than |
| `<=` | less than or equal to | `>=` | greater than or equal to |
| `==` | equal to | `!=` | not equal to |

`and`   `or`   `not`

a comparison using a relational operator is known as a *Boolean expression*, since it evaluates to a *Boolean* (`True` or `False`) value




## EXERCISE: reimplement the `flipCoin` function

- instead of using the `choice` function, use `randint` and an if-else statement

- that is, generate a random integer in range [1, 2]
- if the number is 1, then return "heads
- else, return "tails"

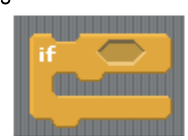


# If statements (cont.)

## you are not required to have an else case to an if statement

- if no else case exists and the test evaluates to false, nothing is done

```
def scale(grade, amount):
    newGrade = grade + amount
    if newGrade > 100:
        newGrade = 100
    return newGrade
```




## an if statement (with no else case) is a 1-way conditional

- depending on the test condition, either execute the indented code or don't

## an if-else statement (with else case) is a 2-way conditional

- depending on the test condition, execute one block of indented code or the other

# If examples

## one more revision: wind chill is not intended for temperatures ≥ 50°

- could add a check for temp ≥ 50, then return what? temp?

```
def windChill(temp, wind):
    if temp >= 50 or wind <= 3:
        return temp
    else:
        chill = 35.74 + 0.6215*temp + \
                (0.4275*temp - 35.75)*(wind**0.16)
        return chill
```

## really want to signify that the value is undefined

- the `float` function will convert a string into its corresponding number

  e.g., `float("12.5")` → `12.5`
- the expression `float("nan")` returns a special value, `nan`, that stands for 'not a number'
- whenever `nan` appears in an expression, the result is still `nan`

```
>>> x = float('nan')
>>> x
nan
>>> x + 1
nan
>>> y = 2 * x - 5
>>> y
nan
```

11

# Cascading if-else

## now have 3 different cases, so need a 3-way conditional

- can accomplish this by nesting if-else statements
- known as a *cascading if-else* (control cascades down from one test to the next)

```
def windChill(temp, wind):
    if temp >= 50:
        return float('nan')
    else:
        if wind <= 3:
            return temp
        else:
            chill = 35.74 + 0.6215*temp + \
                    (0.4275*temp - 35.75)*(wind**0.16)
            return chill
```

## reminder: Python uses indentation to determine code structure

- must make sure to align statements inside the appropriate if-else case

12

# Cascading if-else: elif

because multi-way conditionals are fairly common, a variant exists to simplify the structure

- `elif` is shorthand for else-if
- introduces the next case without having to nest

```
def windChill(temp, wind):
    if temp >= 50:
        return float('nan')
    else:
        if wind <= 3:
            return temp
        else:
            chill = 35.74 + 0.6215*temp + \
                    (0.4275*temp - 35.75)*(wind**0.16)
            return chill
```

```
def windChill(temp, wind):
    if temp >= 50:
        return float('nan')
    elif wind <= 3:
        return temp
    else:
        chill = 35.74 + 0.6215*temp + \
                (0.4275*temp - 35.75)*(wind**0.16)
        return chill
```



# Exercise: letter grades

define a Python function named `letterGrade`, that takes one input (a course average) and returns the corresponding letter grade

- assume grades of "A", "B", "C", "D", and "F" (no + or -)
- assume standard grade cutoffs

  e.g., `letterGrade(90)` should return "A"
  `letterGrade(89)` should return "B"

```
def letterGrade(average):
    ????
```

# Repetition

an if statement provides for conditional execution

- can make a choice between alternatives, choose which (if any to execute)

if we want to repeatedly execute a block of code, need a loop

- loops can be counter-driven
  e.g., roll a die 10 times


- loops can be condition-driven
  e.g., roll dice until doubles


the simplest type of Python loop is a *counter-driven* for loop

```
for i in range(NUM_REPS):
   STATEMENTS_TO_BE_REPEATED
```



# For loop examples

```
>>> for i in range(5):
        print "Howdy"


Howdy
Howdy
Howdy
Howdy
Howdy
```

```
>>> for i in range(5):
        print flipCoin()


tails
tails
tails
heads
tails
```

```
>>> for i in range(10):
        roll = rollDie(6) + rollDie(6)
        print roll


2
9
4
7
4
3
6
7
2
6
```

# Exercise: sum the dice rolls

## suppose we wanted to define a function to sum up dice rolls

- need to initialize a variable to keep track of the sum (starting at 0)
- inside the loop, add each roll to the sum variable
- when done with the loop, display the sum

```
def sumRolls(numRolls):
    ???
```

## similarly, suppose we wanted to average the dice rolls

- calculate the sum, as before
- return sum divided by the number of rolls

```
def avgRolls(numRolls):
    ???
```

17

# Loops & counters

## for loops can be combined with if statements

- common pattern: perform multiple repetitions and count the number of times some event occurs
- e.g., flip a coin and count the number of heads
- e.g., roll dice and count the number of doubles
- e.g., traverse an employee database and find all employees making > $100K

```
def countHeads(numFlips):
    numHeads = 0
    for i in range(numFlips):
        flip = flipCoin()
        if flip == "heads":
            numHeads = numHeads + 1
    return numHeads
```

```
def countDoubles(numRolls):
    numDoubles = 0
    for i in range(numRolls):
        roll1 = rollDie(6)
        roll2 = rollDie(6)
        if roll1 == roll2:
            numDoubles = numDoubles + 1
    return numDoubles
```

18

# Shorthand assignments

a variable that is used to keep track of how many times some event occurs is known as a *counter*

- a counter must be initialized to 0, then incremented each time the event occurs

```
def countHeads(numFlips):
    numHeads = 0
    for i in range(numFlips):
        flip = flipCoin()
        if flip == "heads":
            numHeads += 1
    return numHeads

def countDoubles(numRolls):
    numDoubles = 0
    for i in range(numRolls):
        roll1 = rollDie(6)
        roll2 = rollDie(6)
        if roll1 == roll2:
            numDoubles += 1
    return numDoubles
```

shorthand notation

`number += 1` is equivalent to
`number = number + 1`

`number -= 1` is equivalent to
`number = number - 1`

other shorthand assignments can be used for updating variables

`number += 5` is equivalent to
`number = number + 5`

`number *= 2` is equivalent to
`number = number * 2`




# While loops

the other type of repetition in Python is the *condition-driven* while loop

- similar to an if statement, it is controlled by a Boolean test
- unlike an if, a while loop *repeatedly* executes its block of code as long as the test is true

```
while TEST_CONDITION:
    STATEMENTS_TO EXECUTE_AS_LONG_AS_TEST_IS_TRUE
```

```
>>> flip = flipCoin()
>>> while flip != "heads":
        print flip
        flip = flipCoin()
```

```
>>> roll1 = rollDie(6)
>>> roll2 = rollDie(6)
>>> while roll1 != roll2:
        print roll1, roll2
        roll1 = rollDie(6)
        roll2 = rollDie(6)
```

## Example: hailstone sequence

interesting problem from mathematics

- start a sequence with some positive integer N
- if that number is even, the next number in the sequence is N/2;
  if that number is odd, the next number in the sequence is 3N+1

5 → 16 → 8 → 4 → 2 → 1 → 4 → 2 → 1 → …

15 → 46 → 23 → 70 → 35 → 106 → 53 → 160 → 80 → 40 → 20 → 10
↓
… ← 1 ← 2 ← 4 ← 8 ← 16 ← 5

it has been conjectured that no matter what number you start with, you will end up stuck in the 4-2-1 loop

- has been shown for all values <= $20 \times 2^{58} \approx 5.764 \times 10^{18}$
- but has not been proven to hold in general

21

## Generating a hailstone sequence

need to be able to distinguish between even and odd numbers

- recall the remainder operator, %
- (x % y) evaluates to the remainder after dividing x by y
- thus, (x % 2) evaluates to 0 if x is even, 1 if x is odd

```
def hailstone(num):
    print num
    while num != 1:
        if num%2 == 0:
            num = num / 2
        else:
            num = 3*num + 1
        print num
```

EXERCISE: modify so that it also prints the length of the sequence

22

## Beware of "black holes"

since while loops repeatedly execute as long as the loop test is true, infinite loops are possible (a.k.a. *black hole* loops)

```
def flipUntilHeads():
    flip = flipCoin()
    numFlips = 1
    print numFlips, ":", flip
    while flip != "heads":
        numFlips += 1
        print numFlips, ":", flip
```

PROBLEM?

- a necessary condition for loop termination is that some value relevant to the loop test must change inside the loop

  in the above example, `flip` doesn't change inside the loop

  → if the test succeeds once, it succeeds forever!

- is it a sufficient condition? that is, does changing a variable from the loop test guarantee termination?

  NO – "With great power comes great responsibility."

  fix to above function?

23

## Example: Pig

Pig is a 2-player dice game in which the players take turns rolling a die.

On a given turn, a player rolls until either

1. he/she rolls a 1, in which case his/her turn is over and no points are awarded, or
2. he/she chooses to hold, in which case the sum of the rolls from that player's turn are added to his/her score.

The winner of the game is the first player to reach 100 points.

for example:

SCORE = 0 to start
TURN 1: rolls 5, 2, 4, 6, holds → SCORE = 0 + 17 = 17
TURN 2: rolls 4, 1, done → SCORE = 17 + 0 = 17
TURN 3: rolls 6, 2, 3, hold → SCORE = 17 + 11 = 28
…

24

## Pig simulation

### we want to simulate Pig to determine the best strategy

- i.e., determine the optimal cutoff such that you should keep rolling until the score for a round reaches the cutoff, then hold
- i.e., what is the optimal cutoff that minimizes the expected number of turns

```
def pigTurn(cutoff):
    """Simulates a turn of the game Pig, with the player
       repeatedly rolling until the get a 1 or their score
       reaches the specified cutoff"""
    score = 0
    roll = 0
    while (roll != 1 and score < cutoff):
        roll = rollDie(6)
        if roll == 1:
            score = 0
        else:
            score += roll
        print roll, "-->", score
```

why is roll set to 0 before the loop?
why not set it to rollDie(6)?

EXERCISE: modify the `pigTurn` function so that it returns the score for the round (as opposed to printing rolls/scores)

25

## Pig simulation (cont.)

### EXERCISE: define a `pigGame` function that simulates a Pig game

- has 1 input, the cutoff value for each turn
- it repeatedly calls the `pigTurn` function, totaling up the score for each turn (and displaying the turn # and updated score)
- it stops when the score total reaches 100

```
>>> pigGame(15)
Turn 1 : 19
Turn 2 : 19
Turn 3 : 19
Turn 4 : 19
Turn 5 : 36
Turn 6 : 51
Turn 7 : 71
Turn 8 : 86
Turn 9 : 102
```

```
>>> pigGame(15)
Turn 1 : 16
Turn 2 : 33
Turn 3 : 33
Turn 4 : 33
Turn 5 : 33
Turn 6 : 48
Turn 7 : 68
Turn 8 : 86
Turn 9 : 86
Turn 10 : 86
Turn 11 : 101
```

```
>>> pigGame(15)
Turn 1 : 15
Turn 2 : 15
Turn 3 : 15
Turn 4 : 15
Turn 5 : 15
Turn 6 : 15
Turn 7 : 31
Turn 8 : 31
Turn 9 : 31
Turn 10 : 31
Turn 11 : 31
Turn 12 : 48
Turn 13 : 65
Turn 14 : 65
Turn 15 : 83
Turn 16 : 83
Turn 17 : 99
Turn 18 : 115
```

26

## Pig simulation (cont.)

what can we conclude from running several experiments?

- Simulation 1: a cutoff of 15 yields a game of 12 turns
- Simulation 2: a cutoff of 20 yields a game of 14 turns
- can we conclude that a cutoff of 15 is *better* than a cutoff of 20?

note: because of the randomness of the die, there can be wide variability in the simulations

- note: a single roll of a die is unpredictable
- however: given a *large* number of die rolls, the distribution of the rolls can be predicted (since each die face is equally likely, each should appear ~ 1/6 of time)
- *Law of Large Numbers* states that as the number of repetitions increases to ∞, the percentages should get closer and closer to the expected values

27

## Pig simulation (cont.)

in order to draw reasonable conclusions, will need to perform many experiments and average the results

EXERCISE: modify the `pigGame` function so that it returns the number of turns (as opposed to printing turns/scores)

EXERCISE: define a `pigStats` function that simulates numerous games

- has 2 inputs, the number of games and the cutoff value for each turn
- it repeatedly calls the `pigGame` function the specified number of times, totaling up the number of turns for each game
- it returns the average number of turns over all the games

QUESTION: what is the optimal cutoff that minimizes the number of turns

- how many games do you need to simulate in order to be confident in your answer?

28

## Control summary

### if statements provide for conditional execution

- use when you need to make choices in the code
- control is based on a Boolean (True/False) test
  - 1-way: if (with no else)
  - 2-way: if-else
  - multi-way: cascading if-else, if-elif-elif-…-elif-else

### for loops provide for counter-driven repetition

- use when you need to repeat a task a set number of times
- utilizes the range function (will learn more later)

### while loops provide for conditional repetition

- use when you need to repeat a task but you don't know how many times
- control is based on a Boolean (True/False) test
- as long as test continues to be True, the indented code will be executed
- beware of infinite (black hole) loops

29

## TEST 1: Wednesday, Oct 5

### see syllabus for review sheet

### in addition, consider codingbat.com as a great study resource

- programming practice site run by Nick Parlante at Stanford
- can solve little programming problems, see whether they work

Consider under Python Warmup-1:

- `diff21`
- `near_hundred`
- `sum_double`
- `makes10`

Consider under Python Logic-1:

- `love6`
- `sorta_sum`
- `near10`

30