CSC 221: Computer Programming I

Fall 2011

Intro to Python

- Scratch programming review
- Python & IDLE
- numbers & expressions
- variables & assignments
- strings & concatenation
- functions, parameters, return

Scratch programming review

programming concepts from Scratch

• simple actions/behaviors	(e.g., move, turn, say, play-sound, next-costume)
• control	
• repetition	(e.g., forever, repeat)
• conditional execution	(e.g., if, if-else, repeat-until)
• logic	(e.g., =, >, <, and, or, not)
• arithmetic	(e.g., +, -, *, /)
• sensing	(e.g., touching?, mouse down?, key pressed?)
• variables	(e.g., set, change-by)
• communication/coordination	(e.g., broadcast, when-I-receive)

Python

we are now going to transition to programming in Python

Python is an industry-strength, modern scripting language

- invented by Guido van Rossum in 1989
- has evolved over the years to integrate modern features
  v 2.0 (2000), v 3.0 (2008)
- simple, flexible, free, widely-used, widely-supported, upwardly-mobile

- increasingly being used as the first programming language in college CS programs
- in the real world, commonly used for rapid prototyping
- also used to link together applications written in other languages

The Zen of Python, by Tim Peters

Beautiful is better than ugly. Explicit is better than implicit.
Simple is better than complex. Complex is better than complicated.
Flat is better than nested. Sparse is better than dense.

Readability counts.
Special cases aren't special enough to break the rules. Although practicality beats purity.
Errors should never pass silently. Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.

There should be one -- and preferably only one -- obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.

Now is better than never. Although never is often better than *right* now.

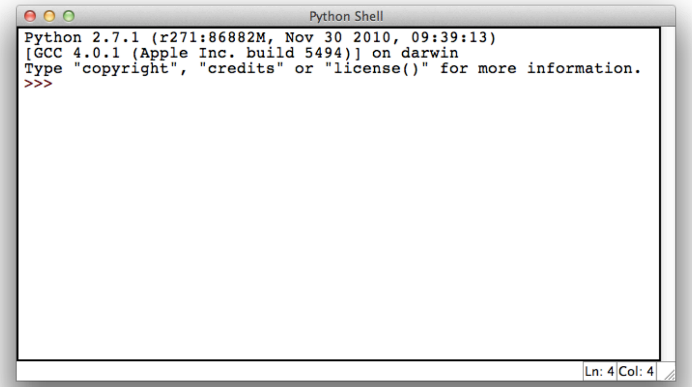
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

Python & IDLE

Python can be freely downloaded from www.python.org

the download includes an Integrated DeveLopment Environment (IDLE) for creating, editing, and interpreting Python programs



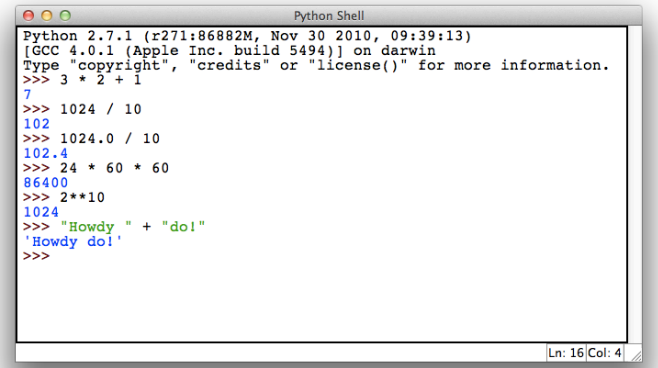
```
Python 2.7.1 (r271:86882M, Nov 30 2010, 09:39:13)
[GCC 4.0.1 (Apple Inc. build 5494)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
```

5

Python interpreter

the >>> prompt signifies that the interpreter is waiting for input

- you can enter an expression to be evaluated or a statement to be executed



```
Python 2.7.1 (r271:86882M, Nov 30 2010, 09:39:13)
[GCC 4.0.1 (Apple Inc. build 5494)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> 3 * 2 + 1
7
>>> 1024 / 10
102
>>> 1024.0 / 10
102.4
>>> 24 * 60 * 60
86400
>>> 2**10
1024
>>> "Howdy " + "do!"
'Howdy do!'
>>>
```

note that IDLE colors text for clarity:

- prompts are reddish
- user input is black (but *strings*, text in quotes, are green)
- answers/results are blue

6

Number types

Python distinguishes between different types of numbers

- **int** integer values, e.g., 2, -10, 1024

  ints are stored using 32 bits, so a range of
  -2^{31} ... $2^{31}-1$ *or* -2,147,483,648 ... 2,147,483,647
  ints can be specified in octal or hexidecimal bases using '0' and '0x' prefixes
  023 → 23_8 → 19_{10} 0x1A → $1A_{16}$ → 26_{10}

- **long** unlimited size integers, e.g., 2L, -10L, 9999999999999999999L

  if you enter or calculate an integer too large to store as an int, the interpreter will automatically convert it to a long (and place an 'L' at the end)

- **float** floating point (real) values, e.g., 3.14, -2.0, 1.9999999

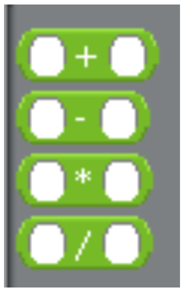
  scientific notation can be used to make very small/large values clearer
  1.234e2 → 1.234×10^2 → 123.4 9e-5 → 9×10^{-5} → 0.00009

- **complex** complex numbers (WE WILL IGNORE)

7

Numbers & expressions

recall in Scratch:



standard numeric operators are provided

`+`	addition, e.g.,	`2 + 3 → 5`	`2 + 3.5 → 5.5`
`-`	subtraction, e.g.,	`10 – 2 → 8`	`99 – 99.5 → -0.5`
`*`	multiplication, e.g.,	`2 * 10 → 20`	`2 * 0.5 → 1.0`
`/`	division, e.g.,	`10 / 3 → 3`	`10.0 / 3 → 3.333…`
`%`	remainder, e.g.,	`10 % 3 → 1`	`10.5 % 2 → 0.5`
`**`	exponent, e.g.,	`2**10 → 1024`	`9**0.5 → 3.0`

note: when an operator is applied to an integer and a real, the result is a real

note: when an operator is applied to two integers, the result is an integer

BE VERY CAREFUL WITH INTEGER DIVISION

8

Exercise: calculations

use IDLE to determine

- the number of seconds in a day
- the number of seconds in a year (assume 365 days in a year)
- your age in seconds

- the number of inches in a mile
- the distance to the sun in inches

- the solution to the Wheat & Chessboard problem

Python strings

in addition to numbers, Python provides a type for representing text

- a *string* is a sequence of characters enclosed in quotes
- Python strings can be written using either single or double quotes (but they must match)

OK:

```
"Creighton"   'Billy Bluejay'   "Bob's your uncle"
```

not OK:

```
"Creighton'    'Bob's your uncle'
```

can nest quotes in a string using the escape '\' character

```
'Bob\'s your uncle'     "He said \"Hi\" to her."
```

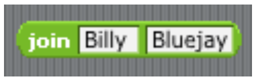
Python strings (cont.)

on rare occasions when you want a string that spans multiple lines, use triple-double quotes

```
"""This is
a single string"""
```

the + operator, when applied to strings, yields concatenation

```
"Billy " + "Bluejay" → "Billy Bluejay"
```

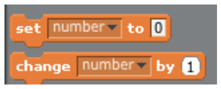


11

Variables & assignments

recall from Scratch, variables were used to store values that could be accessed & updated

- e.g., the name & punch line for the knock-knock joke
- e.g., the number of spins and player bankroll for slots



similarly, in Python can create a variable and assign it a value

- the variable name must start with a letter, consist of letters, digits & underscores (note: no spaces allowed)
- an assignment statement uses '=' to assign a value to a variable
- general form: `VARIABLE = VALUE_OR_EXPRESSION`

```
age = 20

secondsInDay = 24 * 60 * 60

secondsInYear = 365 * secondsInDay

name = "Prudence"

greeting = "Howdy " + name
```

12

Variable names

some reserved words cannot be used for variable names

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

Python libraries tend to use underscores for multi-word variable names

```
first_name    number_of_sides    feet_to_meters
```

we will utilize the more modern (and preferred) camelback style

```
firstName     numberOfSides      feetToMeters
```

note: capitalization matters, so `firstName ≠ firstname`

13

Exercise: knock-knock revisited

suppose we wanted to display a knock-knock joke (as in Scratch)

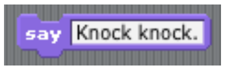
- could use variables to store the name and punch line (as in Scratch)
- could then construct the joke using those values

```
>>> name = "Mike"
>>> punchLine = "My key won't work."
>>> "Knock-knock.  Who's there? " + name + ". " + name + " who? " + punchLine
"Knock-knock.  Who's there? Mike. Mike who? My key won't work."
```

UGLY!

- there is no way to format the text, insert line breaks, etc.
- Python provides a PRINT statement, that allows us to display a result and control its appearance (note: the '`\n`' character produces a new line)
- general form: `print VALUE1, VALUE2, …, VALUEn`

```
>>> print "Knock-knock.\nWho's there?\n", name, "\n", name, "who?\n", punchLine
Knock-knock.
Who's there?
Mike
Mike who?
My key won't work.
```

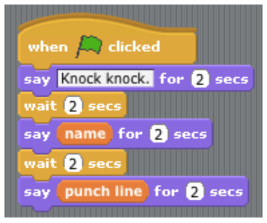


14

Functions

even better solution:

- print each line separately
- group the separate print statements the way Scratch groups together individual blocks into a sequence



in its simplest form, a Python *function* is a sequence of statements grouped together into a block

- general form:
```
def FUNC_NAME():
    SEQUENCE_OF_STATEMENTS
```

- note: the statements that make up the function block must be indented

```
def joke():
    name = "Mike"
    punchLine = "My key won't work"
    print "Knock-knock."
    print "Who's there?"
    print name
    print name, "who?"
    print punchLine
```

15

IDLE editor

we could enter a function definition directly into the interpreter shell

- no way to go back and fix errors
- no way to save the function for later use

instead, can use the file editor built-in to IDLE

- under the File menu, select New Window
- this will open a simple file editor
- enter the function definition (or multiple function definitions)
- save the file (using File→Save) then load the function (using Run→Run Module)
- the function is now defined and can be called in the interpreter shell

- general form of function call: `FUNC_NAME()`

```
>>> joke()
Knock-knock.
Who's there?
Mike
Mike who?
My key won't work
```

16

Exercise: functions

working with your neighbor:

- enter the joke function in the IDLE file editor
- call the function to display the joke

- now, go back to the function and change the name & punch line
- call the function to display the new joke

similarly, define a function for displaying a verse of the bus song:

- have variables for the bus part (e.g., "wheels") & its action (e.g., "round and round")
- then, display the corresponding verse

```
>>> busVerse()
The wheels on the bus go round and round
  round and round
  round and round
The wheels on the bus go round and round
  All through the town.
```

17

Functions with parameters

better solution – parameterize the function

- if the behavior of the function is to depend upon possibly changing values, then add variables inside the parentheses (known as *parameters*)

```
def joke(name, punchLine):
    print "Knock-knock."
    print "Who's there?"
    print name
    print name, "who?"
    print punchLine
```

when the function is called, input values are listed in the parentheses

- values are assigned to the parameters in order listed
- those values can be accessed & used within the function

```
>>> joke("Cash", "No thanks, but I would like some peanuts.")
Knock-knock.
Who's there?
Cash
Cash who?
No thanks, but I would like some peanuts.
```

18

Exercise: parameterize

parameterize your joke function

- should be able to call the function with a different name & punch line each time
- the function will display the joke corresponding to those input values

similarly, parameterize your busVerse function:

- have parameters for the bus part & its action
- the function will display the verse corresponding to the input values

```
>>> busVerse("doors", "swish-swish-swish")
The doors on the bus go swish-swish-swish
  swish-swish-swish
  swish-swish-swish
The doors on the bus go swish-swish-swish
  All through the town.
```

19

Built-in functions

Python provides many built-in functions

- mathematically, a function is a mapping from some number of inputs to an output

```
>>> abs(-20)
20
>>> max(27,3)
27
>>> min(27, 3)
3
```

- e.g., the abs function maps a single number to its absolute value
- e.g., the max/min function maps two or more numbers to the maximum/minimum number

note that these functions do not just *print* the output, they *return* it

- that is, a function call can appear in an expression, its value is the output value

```
>>> q1 = 85
>>> q2 = 68
>>> q3 = 96
>>> avg = (q1 + q2 + q3 - min(q1, q2, q3))/2.0
>>> avg
90.5
```

20

Functions that return values

similarly, we can define our own functions that calculate and return values

- a return statement specifies the output value for a function

```
def feetToMeters(numFeet):
    numMeters = numFeet * 0.3048
    return numMeters
```

```
>>> feetToMeters(1)
0.3048
>>> feetToMeters(10)
3.048
>>> feetToMeters(5280)
1609.344
```

the reverse conversion could be similarly defined using the 0.3048 factor

- or, could make use of the existing `feetToMeters` function

```
def metersToFeet(numMeters):
    numFeet = numMeters / 0.3048
    return numFeet
```

```
def metersToFeet(numMeters):
    numFeet = numMeters / feetToMeters(1)
    return numFeet
```

21

Exercise: years vs. seconds

define a function that converts a number of years into the corresponding number of seconds

```
>>> yearsToSeconds(1)
31536000
>>> yearsToSeconds(20)
630720000
```

similarly, define a function that does the opposite conversion

```
>>> secondsToYears(31536000)
1.0
>>> secondsToYears(1000000)
0.031709791983764585
```

22

Exercise: population growth

visit http://www.census.gov/population/www/popclockus.html to get an up-to-date U.S. population estimate

- e.g., 312,235,809 on 9/17/11

according to U.S. Census Bureau:

- one birth every 7 seconds
- one death every 13 seconds
- one naturalized citizen (net) every 43 seconds

define a function named `estimatePop` that takes a single input, a number of years, and returns the estimated population after that period

```
>>> estimatePop(0)
312235809
>>> estimatePop(1)
315048500
>>> estimatePop(100)
593505013L
```

23

Function doc strings

it is considered good programming practice to put a multiline comment at the beginning of every function definition

- it should briefly document the purpose of the function

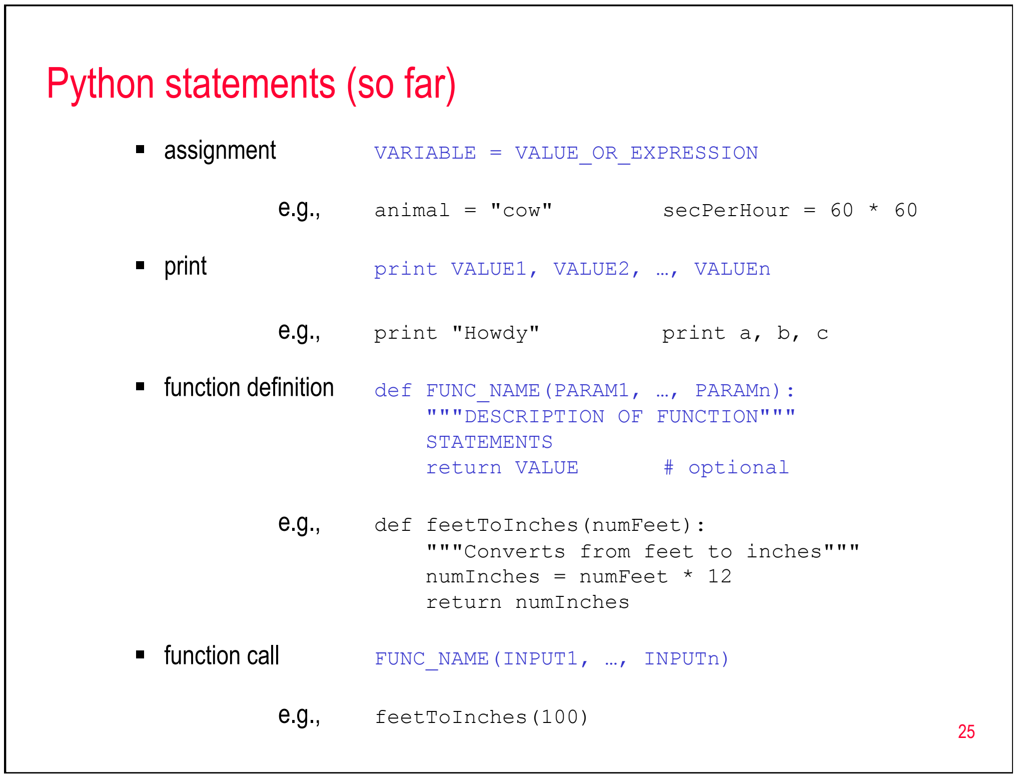
```
def feetToMeters(numFeet):
    """Converts from feet to meters"""
    numMeters = numFeet * 0.3048
    return numMeters
```

- in addition to making it easier to read the code in the editor, the doc string can be viewed in the interpreter shell using the built-in `help` function

```
>>> help(feetToMeters)
Help on function feetToMeters in module __main__:

feetToMeters(numFeet)
    Converts from feet to meters
```

24



Python statements (so far)

- assignment

```
VARIABLE = VALUE_OR_EXPRESSION
```

e.g.,

```
animal = "cow"          secPerHour = 60 * 60
```

- print

```
print VALUE1, VALUE2, …, VALUEn
```

e.g.,

```
print "Howdy"           print a, b, c
```

- function definition

```
def FUNC_NAME(PARAM1, …, PARAMn):
    """DESCRIPTION OF FUNCTION"""
    STATEMENTS
    return VALUE       # optional
```

e.g.,

```
def feetToInches(numFeet):
    """Converts from feet to inches"""
    numInches = numFeet * 12
    return numInches
```

- function call

```
FUNC_NAME(INPUT1, …, INPUTn)
```

e.g.,

```
feetToInches(100)
```

25