CSC 221: Introduction to Programming

Fall 2013

Python data, assignments & turtles

- Scratch programming review
- Python & IDLE
- numbers & expressions
- variables & assignments
- strings & concatenation
- input & output
- turtle graphics

1

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)

2

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

3

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Simple is better than complex.
Flat is better than nested.

Explicit is better than implicit.
Complex is better than complicated.
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.

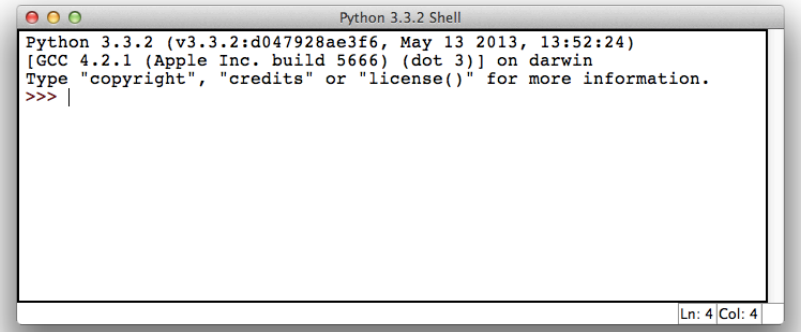
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

4

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

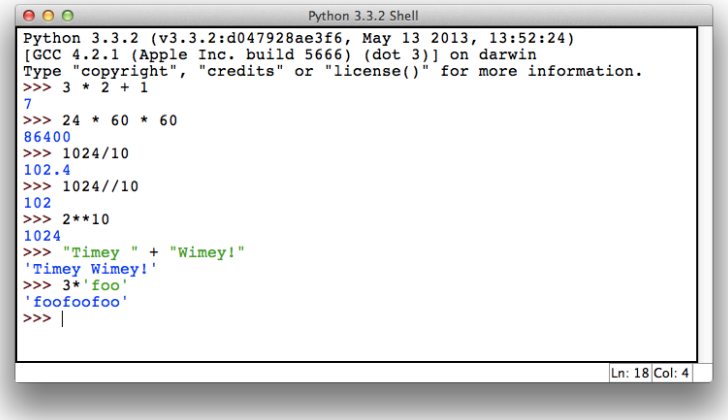


5

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



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, 99999999999999

  ints can be specified in octal or hexidecimal bases using '0o' and '0x' prefixes
  0o23 → 23_8 → 19_{10} 0x1A → $1A_{16}$ → 26_{10}

- **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 x 10^2 → 123.4 9e-5 → 9 x 10^{-5} → 0.00009

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

7

Numbers & expressions

recall in Scratch:

standard numeric operators are provided

`+`	addition	`2+3` → `5`	`2+3.5` → `5.5`
`-`	subtraction	`10-2` → `8`	`99-99.5` → `-0.5`
`*`	multiplication	`2*10` → `20`	`2*0.5` → `1.0`
`/`	division	`10/2.5` → `4.0`	`10/3` → `3.333…`
`**`	exponent	`2**10` → `1024`	`9**0.5` → `3.0`

less common but sometimes useful

`%`	remainder	`10%3` → `1`	`10.5%2` → `0.5`
`//`	integer division	`10//4.0` → `2.5`	`10//4` → `2`

note: spacing around an operator is optional: `2/3 = 2 / 3`

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

9

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-quote symbols (but they must match)

OK:

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

not OK:

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

can nest different quote symbols or use the escape '\' character

```
"Bob's your uncle"          'Bob\'s your uncle'
```

10

Python strings (cont.)

on occasions when you want a string to span multiple lines, use three quotes

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

'''So is
this'''
```

the + operator, when applied to strings, yields concatenation

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

11

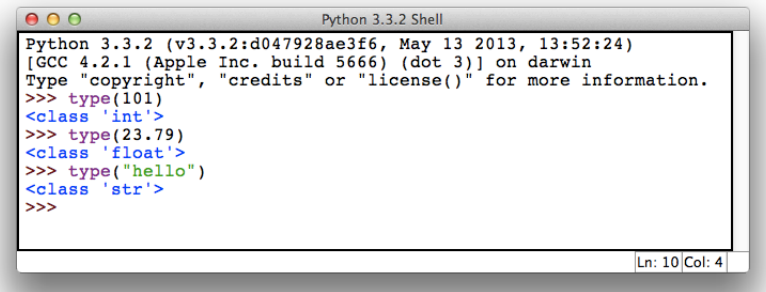
`type` function

in mathematics, a *function* is a mapping from some number of inputs to a single output

- e.g., square root function $\sqrt{9} \rightarrow 3$
- e.g., maximum function $\max(3.8,4.2) \rightarrow 4.2$

Python provides numerous built-in functions for performing useful tasks

- the `type` function takes one *input*, a value, and returns its type as *output*

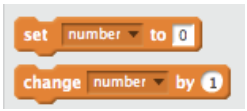


12

Variables & assignments

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

- e.g., the lines spoken by the sprites in conversation
- e.g., the number of times an event occurred



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
```

13

Variable names

some reserved words cannot be used for variable names

and	as	assert	break	class	continue
def	del	elif	else	except	exec
finally	for	from	global	if	import
in	is	lambda	nonlocal	not	or
pass	raise	return	try	while	with
yield	True	False	None		

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`

14

Exercise: population growth

visit http://www.census.gov/popclock for a current U.S. population estimate

- e.g., 316,635,500 at 9:42 pm on 9/8/13

according to U.S. Census Bureau:

- one birth every 8 seconds
- one death every 12 seconds
- one naturalized citizen (net) every 44 seconds

we want to predict the population some number of years in the future

- e.g., what would be the population 10 years from now?

```
>>> 316635500 + (10*365*24*60*60)//8 - (10*365*24*60*60)//12 + (10*365*24*60*60)//44
336942772
```

```
>>> 316635500 + (10*365*24*60*60)//8 - (10*365*24*60*60)//12 \
              + (10*365*24*60*60)//44
336942772
```

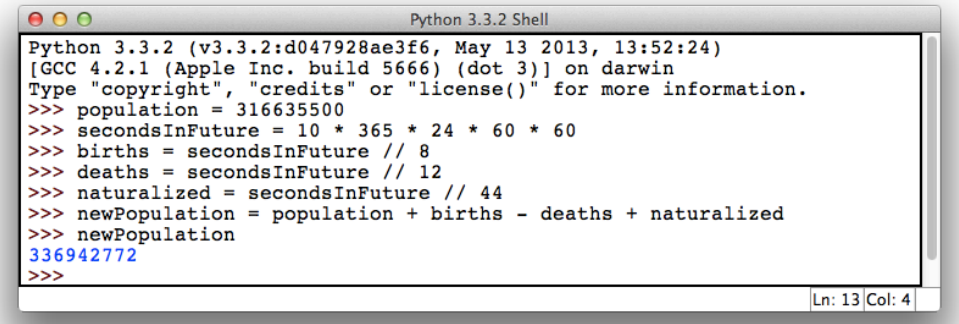
\ is the line continuation character – can break a statement across lines

15

Issues (pt. 1)

- not readable
- significant duplication
- difficult to edit/fix/change

we can address readability & duplication by introducing variables



16

Issues (pt. 2)

variables make the code more readable and reduce duplication, but entering statements directly into the interpreter has limitations

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

better to enter the statements in a separate, editable, re-executable file

the IDLE environment has a built-in file editor

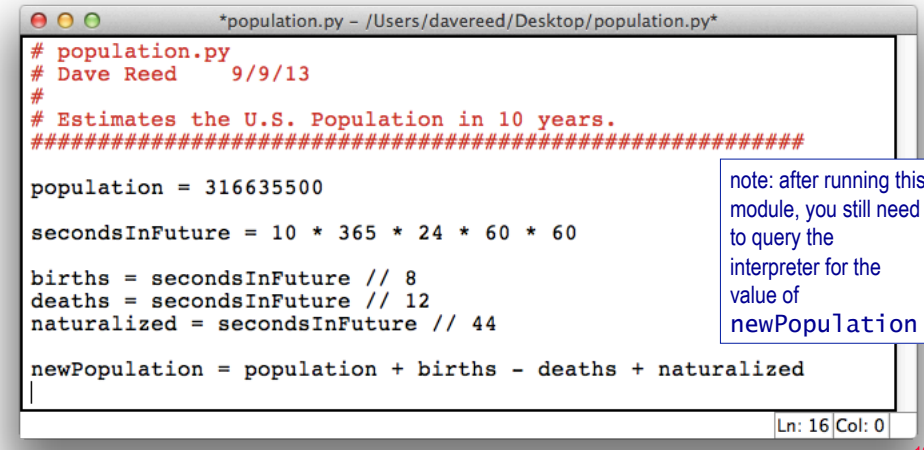
- under the File menu, select New Window
- this will open a simple file editor
- enter the statements
- save the file (using File→Save) then load the function (using Run→Run Module)
- the statements are loaded and run just as if you typed them in directly

17

Modules

a *module* is a file of code that can be repeatedly edited and loaded/run

- lines starting with # are comments, ignored by the Python interpreter
- should start each file with a comment block documenting the file



18

Built-in `print` function

the `print` function can automatically display a value or sequence of values (separated by commas)

```
print(VALUE_1, VALUE_2, …, VALUE_n)
```

population.py – /Users/davereed/Desktop/population.py

```
# population.py
# Dave Reed    9/9/13
#
# Estimates the U.S. Population in 10 years.
#########################################################

population = 316635500

yearsInFuture = 10
secondsInFuture = yearsInFuture * 365 * 24 * 60 * 60

births = secondsInFuture // 8
deaths = secondsInFuture // 12
naturalized = secondsInFuture // 44

newPopulation = population + births - deaths + naturalized

print("After", yearsInFuture, "years, the population would be", \
      newPopulation)
```

Ln: 20 Col: 0

now, a complete message containing the answer is displayed when the module is run

19

User input

as is, changing the `yearsInFuture` assignment and performing a different calculation is easy

- but it would be even better to just ask for the number

the built-in `input` function displays a prompt and reads in the user's input

```
response = input("PROMPT MESSAGE")
```

the `input` function always returns what the user typed as a string

- if you want to treat the input as a number, you must explicitly convert them

```
intResponse = int(input("PROMPT MESSAGE"))

floatResponse = float(input("PROMPT MESSAGE"))
```

20

Final version

now, can enter a different number of years each time the module is run

population.py – /Users/davereed/Desktop/population.py

```
# population.py
# Dave Reed    9/9/13
#
# Estimates the U.S. Population in 10 years.
#############################################################

population = 316635500

yearsInFuture = int(input("How many years in the future? "))
secondsInFuture = yearsInFuture * 365 * 24 * 60 * 60

births = secondsInFuture // 8
deaths = secondsInFuture // 12
naturalized = secondsInFuture // 44

newPopulation = population + births - deaths + naturalized

print("After", yearsInFuture, "years, the population would be", \
      newPopulation)
```

Ln: 20 Col: 0

21

Exercise: reading level

in the 1970's, the U.S. military developed a formula for measuring the readability of text (*for young adults*)

the FORCAST formula assigns a reading age to text based on the following:

$$\text{reading age} = 20 - 15 \times \frac{\#\text{ single syllable words}}{\text{total }\#\text{ of words}}$$

create a Python module that

- prompts the user for the # of single syllable words and total # of words in a text
- calculates the reading age for that text
- prints that reading age (in a readable format)

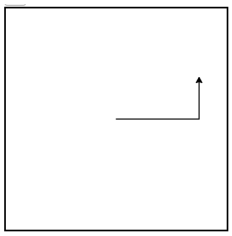
22

Turtle graphics module

IDLE comes with a built-in module for turtle graphics

- must import the module to make the code accessible
- can then call functions from the module via the notation `turtle.FUNCTION()`

```
1 import turtle              # allows us to use the turtles library
2 wn = turtle.Screen()       # creates a graphics window
3 alex = turtle.Turtle()     # create a turtle named alex
4 alex.forward(150)          # tell alex to move forward by 150 units
5 alex.left(90)              # turn by 90 degrees
6 alex.forward(75)           # complete the second side of a rectangle
```



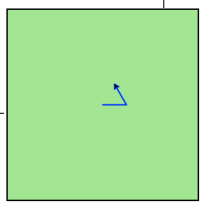
play with the executable code in the interactive book

23

Turtle commands

in addition to movement commands (left, right, forward, backward), there are commands for setting color and pen width

```
1  import turtle
2
3  wn = turtle.Screen()
4  wn.bgcolor("lightgreen")          # set the window background color
5
6  tess = turtle.Turtle()
7  tess.color("blue")                # make tess blue
8  tess.pensize(3)                   # set the width of her pen
9
10 tess.forward(50)
11 tess.left(120)
12 tess.forward(50)
13
14 wn.exitonclick()                  # closes the window
```



complete the exercise – prompt the user for colors

24

For loops

suppose we want to draw a square – need to draw a side & turn (4 times)

```
1  import turtle
2  wn = turtle.Screen()
3  alex = turtle.Turtle()
4
5  alex.forward(50)
6  alex.left(90)
7  alex.forward(50)
8  alex.left(90)
9  alex.forward(50)
10 alex.left(90)
11 alex.forward(50)
12 alex.left(90)
13
14 wn.exitonclick()
```

try the Turtle race lab

a `for loop` specifies a repetitive task

```
for VARIABLE in LIST:
    STATEMENTS
```

```
5 for i in [0, 1, 2, 3]:
6     alex.forward(50)
7     alex.left(90)
```

the `range` function makes this even cleaner

```
for VARIABLE in range(SIZE):
    STATEMENTS
```

```
5 for i in range(4):
6     alex.forward(50)
7     alex.left(90)
```

25