

Introduction to Programming with Python

Variables

- **variable:** A named piece of memory that can store a value.

- Usage:

- Compute an expression's result,
- store that result into a variable,
- and use that variable later in the program.



- **assignment statement:** Stores a value into a variable.

- Syntax:

name = ***value***

- Examples: $x = 5$
 $\text{gpa} = 3.14$

x 5 gpa 3.14

- A variable that has been given a value can be used in expressions.

$x + 4$ is 9

- **Exercise:** Evaluate the quadratic equation for a given a , b , and c .

print

- `print` : Produces text output on the console.

- Syntax:

```
print "Message"
```

```
print Expression
```

- Prints the given text message or expression value on the console, and moves the cursor down to the next line.

```
print Item1, Item2, ..., ItemN
```

- Prints several messages and/or expressions on the same line.

- Examples:

```
print "Hello, world!"
```

```
age = 45
```

```
print "You have", 65 - age, "years until retirement"
```

Output:

```
Hello, world!
```

```
You have 20 years until retirement
```

input

- `input` : Reads a number from user input.

- You can assign (store) the result of `input` into a variable.

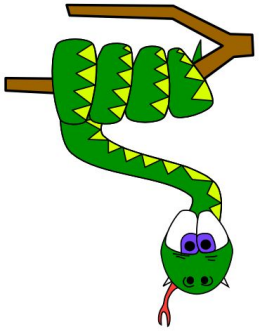
- Example:

```
age = input("How old are you? ")
print "Your age is", age
print "You have", 65 - age, "years until retirement"
```

Output:

```
How old are you? 53
Your age is 53
You have 12 years until retirement
```

- **Exercise:** Write a Python program that prompts the user for his/her amount of money, then reports how many Nintendo Wiis the person can afford, and how much more money he/she will need to afford an additional Wii.



Repetition (loops) and Selection (if/else)

The for loop

- **for loop**: Repeats a set of statements over a group of values.

- Syntax:

```
for variableName in groupOfValues:  
    statements
```

- We indent the statements to be repeated with tabs or spaces.
- **variableName** gives a name to each value, so you can refer to it in the **statements**.
- **groupOfValues** can be a range of integers, specified with the `range` function.

- Example:

```
for x in range(1, 6):  
    print x, "squared is", x * x
```

Output:

```
1 squared is 1  
2 squared is 4  
3 squared is 9  
4 squared is 16  
5 squared is 25
```

range

- The `range` function specifies a range of integers:
 - `range(start, stop)` - the integers between **start** (inclusive) and **stop** (exclusive)
 - It can also accept a third value specifying the change between values.
 - `range(start, stop, step)` - the integers between **start** (inclusive) and **stop** (exclusive) by **step**
 - Example:

```
for x in range(5, 0, -1):  
    print x  
print "Blastoff!"
```

Output:

```
5  
4  
3  
2  
1  
Blastoff!
```
 - **Exercise:** How would we print the "99 Bottles of Beer" song?

Cumulative loops

- Some loops incrementally compute a value that is initialized outside the loop. This is sometimes called a *cumulative sum*.

```
sum = 0
for i in range(1, 11):
    sum = sum + (i * i)
print "sum of first 10 squares is", sum
```

Output:
sum of first 10 squares is 385

- **Exercise:** Write a Python program that computes the factorial of an integer.

if

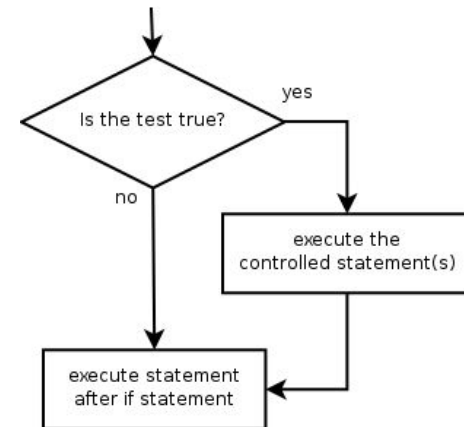
- **if statement:** Executes a group of statements only if a certain condition is true. Otherwise, the statements are skipped.

- Syntax:

```
if condition:  
    statements
```

- Example:

```
gpa = 3.4  
if gpa > 2.0:  
    print "Your application is accepted."
```



if/else

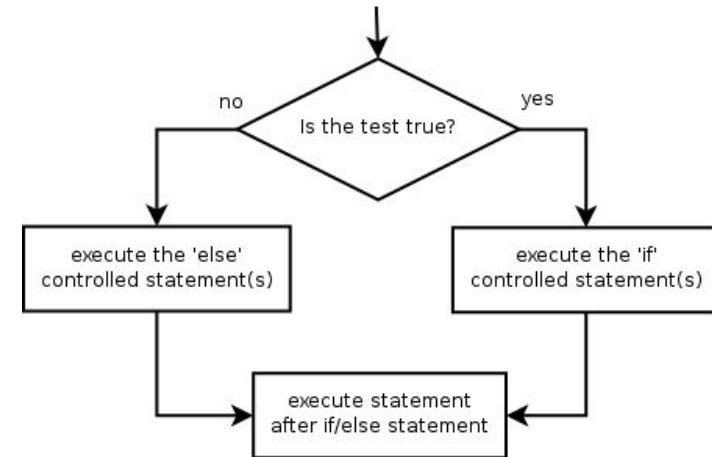
- **if/else statement:** Executes one block of statements if a certain condition is True, and a second block of statements if it is False.

- Syntax:

```
if condition:  
    statements  
else:  
    statements
```

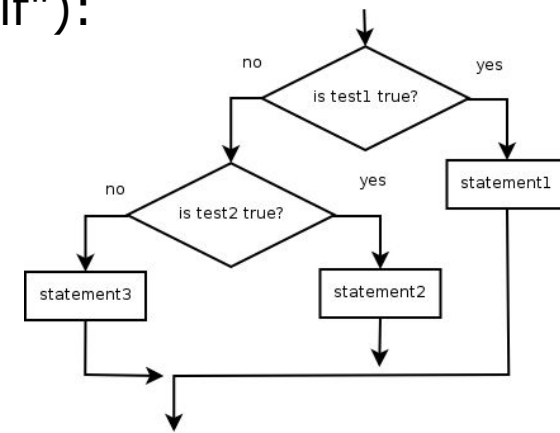
- Example:

```
gpa = 1.4  
if gpa > 2.0:  
    print "Welcome to Mars University!"  
else:  
    print "Your application is denied."
```



- Multiple conditions can be chained with `elif` ("else if"):

```
if condition:  
    statements  
elif condition:  
    statements  
else:  
    statements
```



while

- **while loop:** Executes a group of statements as long as a condition is True.
 - good for *indefinite loops* (repeat an unknown number of times)

- **Syntax:**

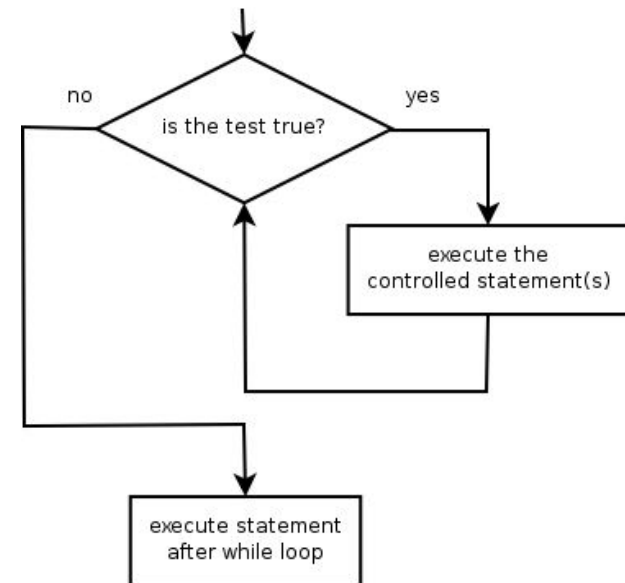
```
while condition:  
    statements
```

- **Example:**

```
number = 1  
while number < 200:  
    print number,  
    number = number * 2
```

- **Output:**

```
1 2 4 8 16 32 64 128
```



Logic

- Many logical expressions use *relational operators*:

Operator	Meaning	Example	Result
==	equals	<code>1 + 1 == 2</code>	True
!=	does not equal	<code>3.2 != 2.5</code>	True
<	less than	<code>10 < 5</code>	False
>	greater than	<code>10 > 5</code>	True
<=	less than or equal to	<code>126 <= 100</code>	False
>=	greater than or equal to	<code>5.0 >= 5.0</code>	True

- Logical expressions can be combined with *logical operators*:

Operator	Example	Result
and	<code>9 != 6 and 2 < 3</code>	True
or	<code>2 == 3 or -1 < 5</code>	True
not	<code>not 7 > 0</code>	False

- Exercise:** Write code to display and count the factors of a number.

Strings

- **string:** A sequence of text characters in a program.

- Strings start and end with quotation mark " or apostrophe ' characters.
- Examples:

```
"hello"
```

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

```
"This, too, is a string.    It can be very long!"
```

- A string may not span across multiple lines or contain a " character.

```
"This is not  
a legal String."
```

```
"This is not a "legal" String either."
```

- A string can represent characters by preceding them with a backslash.

- \t tab character
- \n new line character
- \" quotation mark character
- \\ backslash character

- Example: "Hello\tthere\nHow are you?"

Indexes

- Characters in a string are numbered with *indexes* starting at 0:

- Example:

```
name = "P. Diddy"
```

index	0	1	2	3	4	5	6	7
character	P	.		D	i	d	d	y

- Accessing an individual character of a string:

variableName [***index***]

- Example:

```
print name, "starts with", name[0]
```

Output:

```
P. Diddy starts with P
```

String properties

- `len(string)` - number of characters in a string (including spaces)
- `str.lower(string)` - lowercase version of a string
- `str.upper(string)` - uppercase version of a string

- Example:

```
name = "Martin Douglas Stepp"  
length = len(name)  
big_name = str.upper(name)  
print big_name, "has", length, "characters"
```

Output:

```
MARTIN DOUGLAS STEPP has 20 characters
```

raw_input

- `raw_input` : Reads a string of text from user input.

- Example:

```
name = raw_input("Howdy, pardner. What's yer name? ")  
print name, "... what a silly name!"
```

Output:

```
Howdy, pardner. What's yer name? Paris Hilton  
Paris Hilton ... what a silly name!
```


Text processing

- **text processing:** Examining, editing, formatting text.
 - often uses loops that examine the characters of a string one by one
- A `for` loop can examine each character in a string in sequence.
 - Example:

```
for c in "booyah":  
    print c
```

Output:

```
b  
o  
o  
y  
a  
h
```

Strings and numbers

- `ord(text)` - converts a string into a number.
 - Example: `ord("a")` is 97, `ord("b")` is 98, ...
 - Characters map to numbers using standardized mappings such as *ASCII* and *Unicode*.
- `chr(number)` - converts a number into a string.
 - Example: `chr(99)` is "c"
- **Exercise:** Write a program that performs a rotation cypher.
 - e.g. "Attack" when rotated by 1 becomes "buubdl"

- 
- Any Q?



File processing

- Many programs handle data, which often comes from files.
- Reading the entire contents of a file:

```
variableName = open ("filename") .read()
```

Example:

```
file_text = open("bankaccount.txt").read()
```

Line-by-line processing

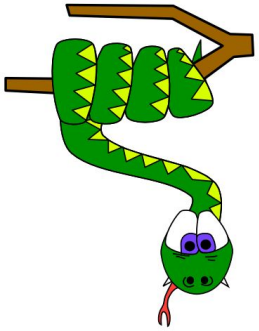
- Reading a file line-by-line:

```
for line in open("filename").readlines():  
    statements
```

Example:

```
count = 0  
for line in open("bankaccount.txt").readlines():  
    count = count + 1  
print "The file contains", count, "lines."
```

- **Exercise:** Write a program to process a file of DNA text, such as:
 ATGCAATTGCTCGATTAG
 - Count the percent of C+G present in the DNA.



Graphics

DrawingPanel

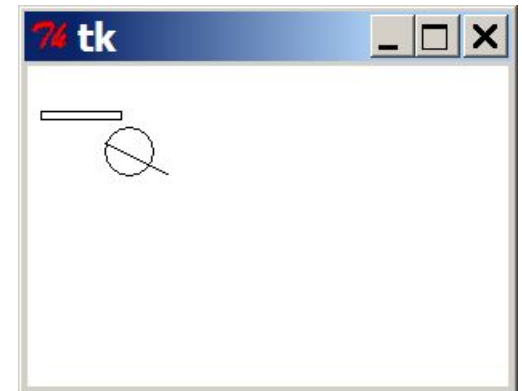
- To create a window, create a `drawingpanel` and its graphical pen, which we'll call `g` :

```
from drawingpanel import *  
panel = drawingpanel(width, height)  
g = panel.get_graphics()  
... (draw shapes here) ...  
panel.mainloop()
```

- The window has nothing on it, but we can draw shapes and lines on it by sending commands to `g` .

- Example:

```
g.create_rectangle(10, 30, 60, 35)  
g.create_oval(80, 40, 50, 70)  
g.create_line(50, 50, 90, 70)
```



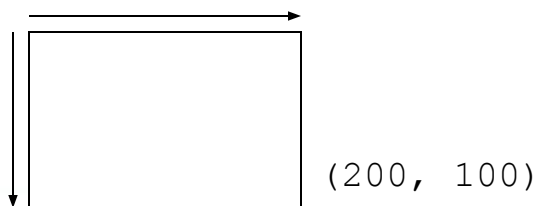
Graphical commands

Command	Description
<code>g.create_line(x1, y1, x2, y2)</code>	a line between (x1 , y1), (x2 , y2)
<code>g.create_oval(x1, y1, x2, y2)</code>	the largest oval that fits in a box with top-left corner at (x1 , y1) and bottom-left corner at (x2 , y2)
<code>g.create_rectangle(x1, y1, x2, y2)</code>	the rectangle with top-left corner at (x1 , y1), bottom-left at (x2 , y2)
<code>g.create_text(x, y, text="text")</code>	the given text at (x , y)

- The above commands can accept optional outline and fill colors.
`g.create_rectangle(10, 40, 22, 65, fill="red", outline="blue")`

- The coordinate system is y-inverted:

(0, 0)



Drawing with loops

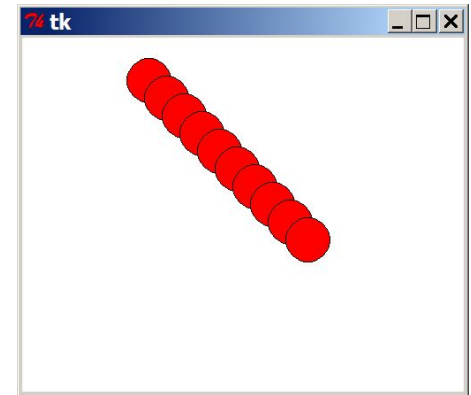
- We can draw many repetitions of the same item at different x/y positions with `for` loops.
 - The x or y assignment expression contains the loop counter, `i`, so that in each pass of the loop, when `i` changes, so does x or y.

```
from drawingpanel import *

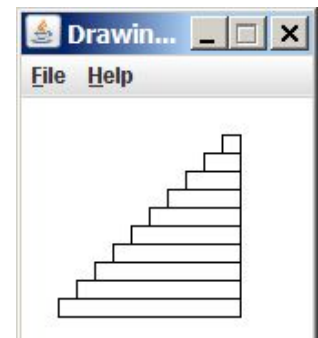
window = drawingpanel(500, 400)
g = window.get_graphics()

for i in range(1, 11):
    x = 100 + 20 * i
    y = 5 + 20 * i
    g.create_oval(x, y, x + 50, y + 50, fill="red")

window.mainloop()
```

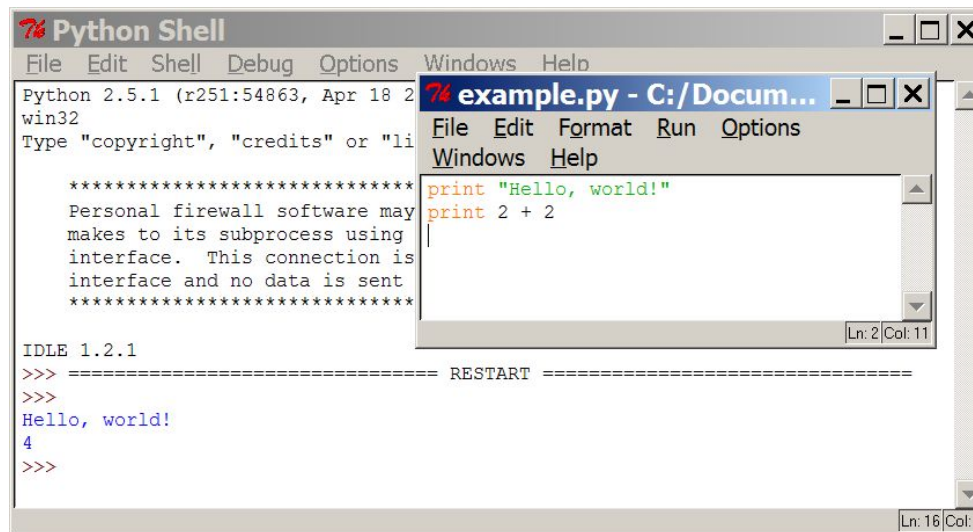


- **Exercise:** Draw the figure at right.



Programming basics

- **code** or **source code**: The sequence of instructions in a program.
- **syntax**: The set of legal structures and commands that can be used in a particular programming language.
- **output**: The messages printed to the user by a program.
- **console**: The text box onto which output is printed.
 - Some source code editors pop up the console as an external window, and others contain their own console window.

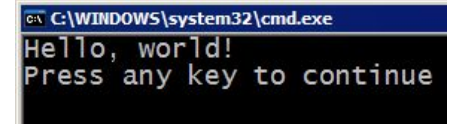


The image shows two overlapping windows. The background window is titled 'Python Shell' and contains the following text:

```
Python 2.5.1 (r251:54863, Apr 18 2006)
win32
Type "copyright", "credits" or "license()" for more
>>>
*****
Personal firewall software may
makes to its subprocess using
interface. This connection is
interface and no data is sent
*****
IDLE 1.2.1
>>> ===== RESTART =====
>>>
Hello, world!
4
>>>
```

The foreground window is titled 'example.py - C:/Docum...' and contains the following code:

```
File Edit Format Run Options
Windows Help
print "Hello, world!"
print 2 + 2
|
Ln: 2 Col: 11
```

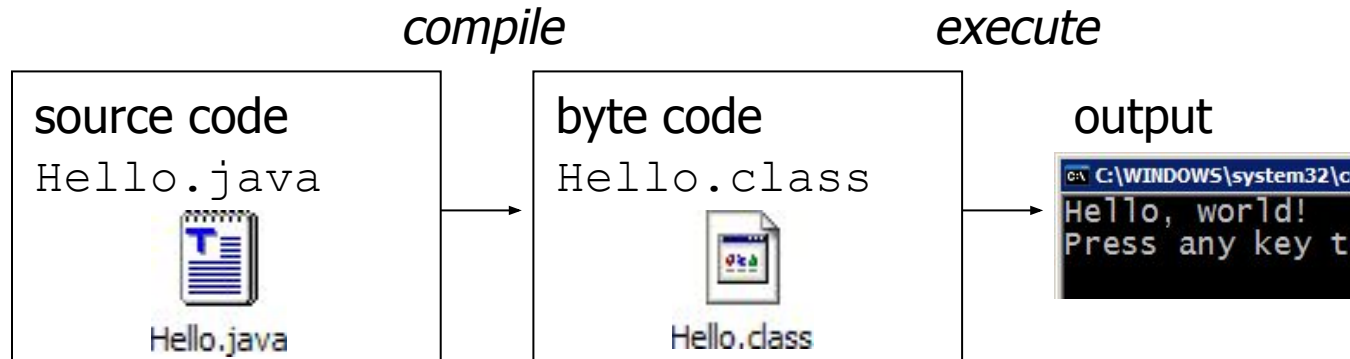


The image shows a command prompt window titled 'C:\WINDOWS\system32\cmd.exe' with the following text:

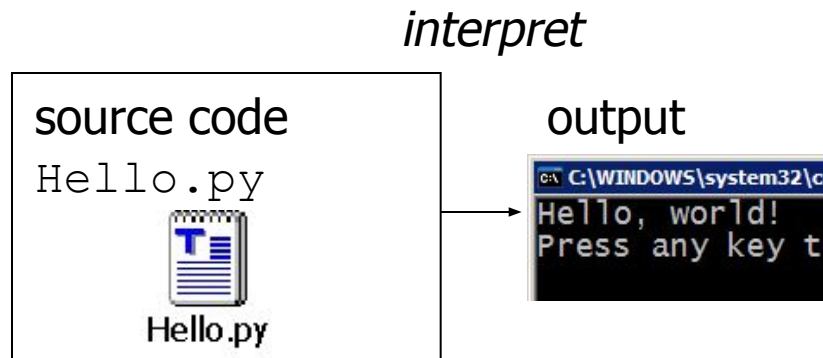
```
Hello, world!
Press any key to continue
```

Compiling and interpreting

- Many languages require you to *compile* (translate) your program into a form that the machine understands.



- Python is instead directly *interpreted* into machine instructions.



Expressions

- **expression:** A data value or set of operations to compute a value.

Examples: $1 + 4 * 3$

42

- Arithmetic operators we will use:

$+$ $-$ $*$ $/$ addition, subtraction/negation, multiplication, division

$\%$ modulus, a.k.a. remainder

$**$ exponentiation

- **precedence:** Order in which operations are computed.

- $*$ $/$ $\%$ $**$ have a higher precedence than $+$ $-$

$1 + 3 * 4$ is 13

- Parentheses can be used to force a certain order of evaluation.

$(1 + 3) * 4$ is 16

Integer division

- When we divide integers with $/$, the quotient is also an integer.

$$\begin{array}{r} 3 \\ 4 \overline{) 14} \\ \underline{12} \\ 2 \end{array}$$

$$\begin{array}{r} 52 \\ 27 \overline{) 1425} \\ \underline{135} \\ 75 \\ \underline{54} \\ 21 \end{array}$$

- More examples:

- $35 / 5$ is 7

- $84 / 10$ is 8

- $156 / 100$ is 1

- The $\%$ operator computes the remainder from a division of integers.

$$\begin{array}{r} 3 \\ 4 \overline{) 14} \\ \underline{12} \\ 2 \end{array}$$

$$\begin{array}{r} 43 \\ 5 \overline{) 218} \\ \underline{20} \\ 18 \\ \underline{15} \\ 3 \end{array}$$

Real numbers

- Python can also manipulate real numbers.

- Examples: `6.022` `-15.9997` `42.0` `2.143e17`

- The operators `+` `-` `*` `/` `%` `**` `()` all work for real numbers.

- The `/` produces an exact answer: `15.0 / 2.0` is `7.5`

- The same rules of precedence also apply to real numbers:
Evaluate `()` before `*` `/` `%` before `+` `-`

- When integers and reals are mixed, the result is a real number.

- Example: `1 / 2.0` is `0.5`

- The conversion occurs on a per-operator basis.

$$\begin{array}{rcl} \underline{7 / 3} * 1.2 + 3 / 2 & & \\ \underline{2} * 1.2 + 3 / 2 & & \\ \underline{2.4} + \underline{3 / 2} & & \\ \underline{2.4} + \underline{1} & & \\ \underline{3.4} & & \end{array}$$

Math commands

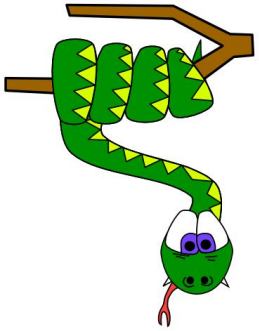
- Python has useful commands for performing calculations.

Command name	Description
<code>abs(value)</code>	absolute value
<code>ceil(value)</code>	rounds up
<code>cos(value)</code>	cosine, in radians
<code>floor(value)</code>	rounds down
<code>log(value)</code>	logarithm, base e
<code>log10(value)</code>	logarithm, base 10
<code>max(value1, value2)</code>	larger of two values
<code>min(value1, value2)</code>	smaller of two values
<code>round(value)</code>	nearest whole number
<code>sin(value)</code>	sine, in radians
<code>sqrt(value)</code>	square root

Constant	Description
e	2.7182818...
pi	3.1415926...

- To use many of these commands, you must write the following at the top of your Python program:

```
from math import *
```



Text and File Processing