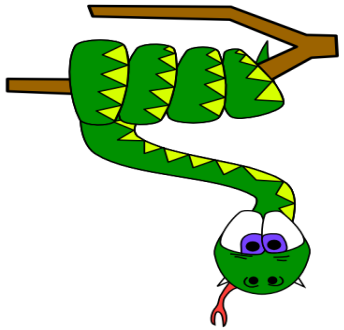


python™

# Basics of Programming with Python*

*These slides are taken and modified from the web sources, especially Marty Stepp Lectures and stackabuse.com web site

# Expressions

- **expression**: A data value or set of operations to compute a value.
  Examples: `1 + 4 * 3`

- 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`

# Operators

- When we divide integers with `/` , the quotient is also an integer.
  - examples:
    - `35 / 5` is `7`
    - `84 / 10` is `8`
    - `156 / 100` is `1`
- The `%` operator computes the remainder from a division of integers.
- `84 % 10= ?`
- 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.

# Math commands

- Python has useful commands for performing calculations.

| Command name | Description |
|---|---|
| `abs(`***value***`)` | absolute value |
| `ceil(`***value***`)` | rounds up |
| `cos(`***value***`)` | cosine, in radians (not degree) |
| `floor(`***value***`)` | rounds down |
| `log(`***value***`)` | logarithm, base *e* |
| `log10(`***value***`)` | logarithm, base 10 |
| `max(`***value1***`, `***value2***`)` | larger of two values |
| `min(`***value1***`, `***value2***`)` | smaller of two values |
| `round(`***value***`)` | nearest whole number |
| `sin(`***value***`)` | sine, in radians |
| `sqrt(`***value***`)` | square root |

# Variables

- **variable**: A named piece of memory that can store a value.
- **assignment statement**: Stores a value into a variable.
  - Syntax:

    ***name*** `=` ***value***

  - Examples:`x = 5`

    `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`

# 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
```

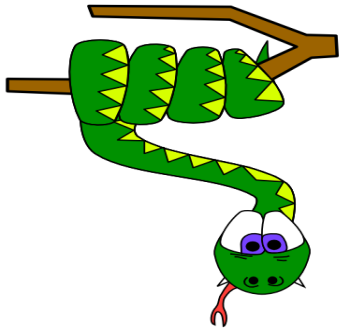
# `input`

- `input` : Reads a number from user input.
  - You can assign (store) the result of `input` into a variable.
  - Example:
    **`Syntax: X = input()`**

```
age = input()
print ("Your age is",age)
```

Output:

```
How old are you? 53
Your age is 53
```




# 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.

- 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 ("Next!")
```

Output:

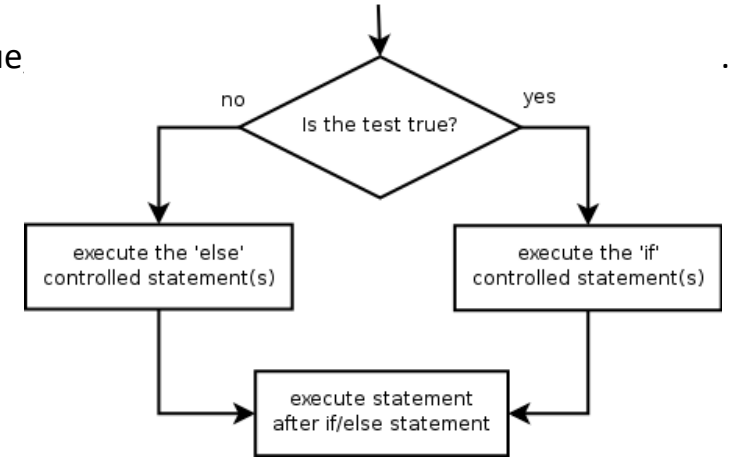
```
5
4
3
2
1
Next!
```

# if/else

- **`if/else` statement**: Executes one block of statements if a certain condition is True

  - Syntax:

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

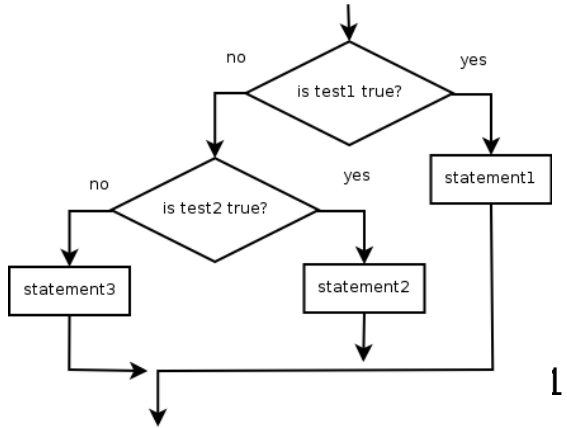



- Example:

```
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:

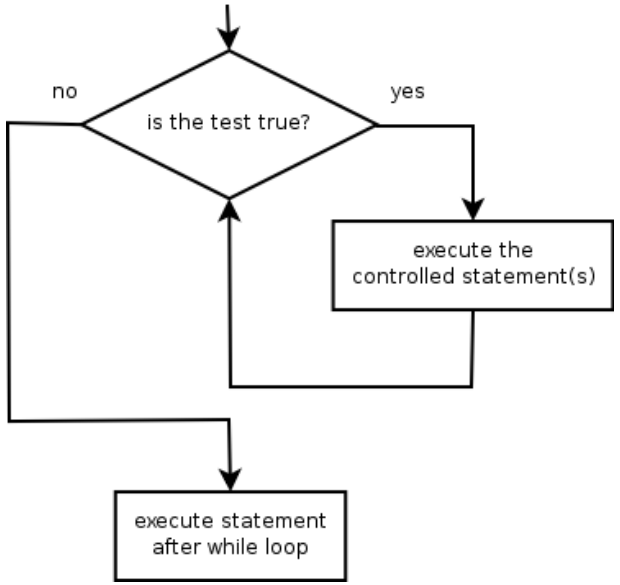
```
while condition:
    statements
```

- Example:

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

  - Output:

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

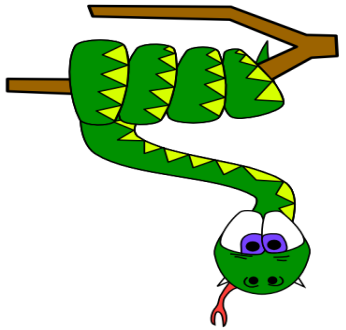


# Logic

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

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

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




# Text and File Processing

# 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:
  - name= “Jean”

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

  Output:

```
Jean starts with J
```

# 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
- `str.replace('c1','c2')` –replace a character by another
- `Str[n1:n2]` – gets "slices" of a string: str[0: 3]

- Example:

```
name = "Jean Alice"
length = len(name)-1
short_name = str.upper(name)
print (short_name, "has", length , "characters")
```

Output ?

# 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?

  text.*count*('n') gives the number of occurrence of n in the text

- `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`"

# 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("BA.txt").readlines():
    count = count + 1
print "The file contains", count, "lines."
```

common explicit methods (read, readline, and readlines) to read in data from a file.
The read method will read in all the data into one text string.

readline which is a useful way to only read in individual line incremental amounts at a time and return them as strings.

The last explicit method, readlines, will read all the lines of a file and return them as a list of strings.