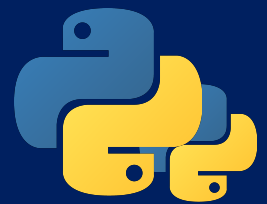


# CSC 322: Computer Organization Lab

Lecture 02: Introduction to Programming with Python

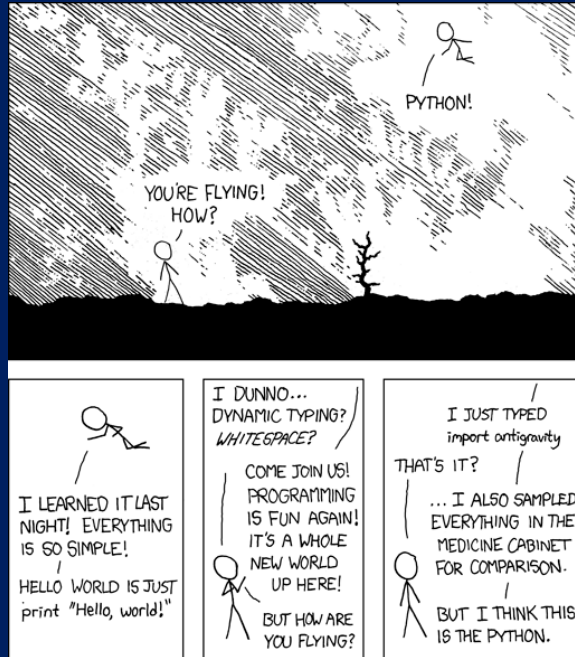


## Outline

- Programming languages and Python
- Basic programs and numeric data
- Control statements
- Text processing



# Why Python?



3

# Languages

- Some powerful ones:
  - C
    - Systems Programming
  - FORTRAN
    - science / engineering
  - COBOL
    - business data
  - LISP
    - logic and AI
  - BASIC
    - a simple language



4

# Python

- Created in 1991 by Guido van Rossum (now at Google)
  - Named for Monty Python
- Useful as a **scripting language**
  - **script**: A program meant for use in small/medium projects
- Used by:
  - Google, Yahoo!, Youtube
  - Many Linux distributions
  - Games and apps (e.g. Eve Online)



5

# Installing Python

## Windows:

- Download Python from <http://www.python.org>
- Install Python.
- Run **Idle** from the Start Menu.

## Mac OS X:

- Python is already installed.
- Open a terminal and run `python` or run Idle from Finder.

## Linux:

- Chances are you already have Python installed. To check, run `python` from the terminal.
- If not, install from your distribution's package system.



6

# Python 2 or Python 3?

*Python 2.x is legacy, Python 3.x is the present and future of the language*

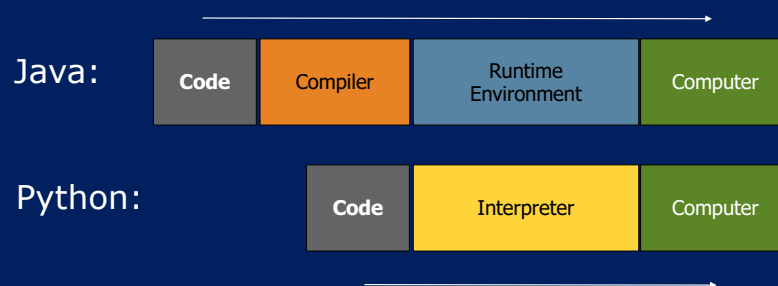


7

# Interpreted Languages

- **interpreted**

- Not compiled like many other languages (Java, C, C++)
- Code is written and then directly executed by an **interpreter**
- Type commands into interpreter and see immediate results



8

# The Python Interpreter

- Allows you to type commands one-at-a-time and see results
- A great way to explore Python's syntax

```
haider — Python — 80x24
yoda:~ haider$ python3
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
>>> 
```



## Basic Programs and Numeric Data



# The `print` Statement

- A Python program's code is just written directly into a file

```
print "Hello, World!"    // Python 2
print ("Hello, World!")  // Python 3
```



11

# The `print` Statement

```
yoda:~ haidar$ python
Python 2.7.10 (default, Jul 15 2017, 17:16:57)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello, World!"
Hello, World!
>>> []
```

```
yoda:~ haidar$ python3
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
>>> []
```



12

# Comments

- Anything after a # is ignored by Python

## swallows2.py

```
1 # Suzy Student, CSE 142, Fall 2097
2 # This program prints important messages.
3 print "Hello, world!"
4 print                # blank line
5 print "Suppose two swallows \"carry\" it together."
6 print 'African or "European" swallows?'
```



13

## An Equivalent to the Main() method

- Python does not have a main() method like Java or C but every module has a special attribute called `__name__`
- A good way to control the entry point to your program and write clean and structured code is to conditionally execute a module:

```
if __name__ == "__main__":
    main()
```

- Example

```
def main():
    print("Hello World")

if __name__ == "__main__":
    main()
```



14

# Expressions

- **expression:** A value or operation(s) to compute a value.

Example: `1 + 4 * 3`

- **Arithmetic operators:**

- `+` `-` `*` `/`      add, subtract/negate, multiply, divide  
- `**`                  exponentiate  
- `%`                    modulus, a.k.a. remainder

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

- `*` `/` `%` `**`      have a higher precedence than `+` `-`

`1 + 3 * 4`      is 13

`(1 + 3) * 4`    is 16



15

# Sentences or Lines

`x = 2` ← *Assignment statement*

`x = x + 2` ← *Assignment with expression*

`print(x)` ← *Print statement*

*Variable*

*Operator*

*Constant*

*Function*



16



# Integer division

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

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

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

- `35 / 5` is 7
- `84 / 10` is 8

- The `%` operator computes a remainder from integer division.

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

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



17

# Variables

- A **variable** is a named place in the memory where a programmer can store data and later retrieve the data using the **variable** "name"

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

– Syntax:

**name = expression**

– Examples: `x = 5`

`gpa = 3.14`

`x` 5

`gpa` 3.14

– A variable can be used in expressions.

`x + 4` is 9



18

## Exercise

- This program's code is redundant. Improve it with variables:

```
print ("Subtotal:")
print (38 + 40 + 30)
print ("Tax:")
print ((38 + 40 + 30) * .09)
print ("Tip:")
print ((38 + 40 + 30) * .15)
print ("Total:")
print (38 + 40 + 30 + (38 + 40 + 30) * .15 + (38 + 40 + 30) * .09)
```



19

## Data Types

- **type:** A category or set of data values.
  - Constrains the operations that can be performed on the data
  - Examples: integer, real number, text string
- Python is relaxed about types.
  - A variable's type does not need to be declared.
  - A variable can change types as a program is running.

Value	Python type
42	int
3.14	float
"nil"	str



20

# Parameters

- **parameter:** A value supplied to a command as you run it.

– Syntax:

**command** ( **value** )

**command** ( **value**, **value**, ..., **value** )

- Example:

```
import math    // more about this one later!

print (math.sqrt(25))
print (math.sqrt(15 + 10 * 10 + 6))
x = 5
print (math.sqrt(x + sqrt(16)))
```



21

# Parameters

- **parameter:** A value supplied to a command as you run it.

– Syntax:

**command** ( **value** )

**command** ( **value**, **value**, ..., **value** )

- Example:

```
from math import *    // more about this one later!

print (sqrt(25))
print (sqrt(15 + 10 * 10 + 6))
x = 5
print (sqrt(x + sqrt(16)))
```



22

# Math commands

Function name	Description	Constant	Description
<code>abs (value)</code>	absolute value	<code>e</code>	2.7182818...
<code>ceil (value)</code>	rounds up	<code>pi</code>	3.1415926...
<code>cos (value)</code>	cosine, in radians		
<code>floor (value)</code>	rounds down		
<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		

- To use these commands, place this line atop your program:

```
from math import *
```



23

# input

- `input` : Reads a string from the user's keyboard.
  - You can store the result of `input` into a variable.

– Example:

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

Output:

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



24

# Converting User Input

- If we want to read a number from the user, we must convert it from a string to a number using a type conversion function
- Later we will deal with bad input data

```
inp = input('Europe floor?')
usf = int(inp) + 1
print('US floor', usf)
```



25

## raw\_input

- Similar to `input` in Python 3 => Reads a string of text from the user's keyboard.

– Example:

```
name = raw_input("Howdy. What's yer name? "). // Python 2
print name, "... what a silly name!"
```

Output:

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



26

# Control Statements



## Comparison Operators

- Boolean expressions ask a question and produce a Yes or No result which we use to control program flow
- Boolean expressions using comparison operators evaluate to True / False or Yes / No
- Comparison operators look at variables but do not change the variables



# Indentation

- Increase indent after an if statement or for statement (after : )
- Maintain indent to indicate the scope of the block (which lines are affected by the if/for)
- Reduce indent back to the level of the if statement or for statement to indicate the end of the block
- Blank lines are ignored - they do not affect indentation
- Comments on a line by themselves are ignored with regard to indentation



29

## Warning: Do Not Mix Tabs and spaces

- Some editors automatically use spaces for files with ".py" extension (nice!)
- Most text editors can turn tabs into spaces
  - Make sure to enable this feature
- Python cares a *lot* about how far a line is indented. If you mix tabs and spaces, you may get "indentation errors" even if everything looks fine



30

*increase / maintain* after if or for  
*decrease* to indicate end of block

```
x = 5
if x > 2 :
    print('Bigger than 2')
    print('Still bigger')
print('Done with 2')

for i in range(5) :
    print(i)
    if i > 2 :
        print('Bigger than 2')
    print('Done with i', i)
print('All Done')
```



31

## if

- **if statement:** Executes a set of commands only if a certain condition is True. Otherwise, the commands are skipped.

– Syntax:

```
if condition :
    statements
```

– Example:

```
gpa = input("What is your GPA? ")
if gpa > 2.0:
    print "Your application is accepted."
```



32



# if/else

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

– Syntax:

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

– Example:

```
gpa = input("What is your GPA? ")  
if gpa > 2.0:  
    print "Welcome to Mars University!"  
else:  
    print "Your application is denied."
```

- Multiple conditions can be chained with `elif`



33

# 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

– Logical expressions can be combined using *logical operators*:

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



34

# Using in as a Logical Operator

- The in keyword can also be used to check to see if one string is "in" another string
- The in expression is a logical expression that returns True or False and can be used in an if statement

```
>>> fruit = 'banana'
>>> 'n' in fruit
True
>>> 'm' in fruit
False
>>> 'nan' in fruit
True
>>> if 'a' in fruit :
...     print('Found it!')
...
Found it!
>>>
```



35

# String Library

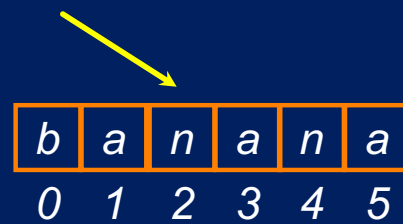
```
>>> stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
['capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartmention', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
```



36

## Searching a String

- We use the `find()` function to search for a substring within another string
- `find()` finds the first occurrence of the substring
- If the substring is not found, `find()` returns `-1`
- Remember that string position starts at zero



```
>>> fruit = 'banana'
>>> pos = fruit.find('na')
>>> print(pos)
2
>>> aa = fruit.find('z')
>>> print(aa)
-1
```



37

## Stripping Whitespace

- Sometimes we want to take a string and remove whitespace at the beginning and/or end
- `lstrip()` and `rstrip()` remove whitespace at the left or right
- `strip()` removes both beginning and ending whitespace

```
>>> greet = ' Hello Bob '
>>> greet.lstrip()
'Hello Bob '
>>> greet.rstrip()
' Hello Bob'
>>> greet.strip()
'Hello Bob'
>>>
```



38

## Prefixes

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```



39

## Parsing and Extracting

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

21 ↓                      31 ↓

uct.ac.za

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ', atpos)
>>> print(sppos)
31
>>> host = data[atpos+1 : sppos]
>>> print(host)
uct.ac.za
```



40

# for loops

```
for name in range(start, end):  
    statements
```

```
for name in range(start, end, step):  
    statements
```

– Repeats for values **start** (inclusive) to **end** (exclusive)

```
>>> for i in range(2, 6):  
    print i  
2  
3  
4  
5  
>>> for i in range(15, 0, -5):  
    print i, "squared is", (i * i)  
15 squared is 225  
10 squared is 100  
5 squared is 25
```



41

# Cumulative loops

- Some loops incrementally compute a value.
  - sometimes called a *cumulative* loop

```
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
```

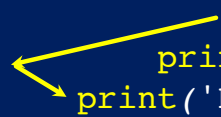


42

# Breaking Out of a Loop

- The break statement ends the current loop and jumps to the statement immediately following the loop
- It is like a loop test that can happen anywhere in the body of the loop

```
while True:
    line = input('> ')
    if line == 'done' :
        break
    print(line)
print('Done!')
```



```
> hello there
hello there
> finished
finished
> done
Done!
```

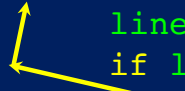


43

# Finishing an Iteration with continue

- The continue statement ends the current iteration and jumps to the top of the loop and starts the next iteration

```
while True:
    line = input('> ')
    if line[0] == '#' :
        continue
    if line == 'done' :
        break
    print(line)
print('Done!')
```



```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```



44

## Looping Through a Set

```
print('Before')
for thing in [9, 41, 12, 3, 74, 15] :
    print(thing)
print('After')
```

```
$ python basicloop.py
```

```
Before
```

```
9
```

```
41
```

```
12
```

```
3
```

```
74
```

```
15
```

```
After
```



45

## Finding the Largest Value

```
largest_so_far = -1
print('Before', largest_so_far)
for the_num in [9, 41, 12, 3, 74, 15] :
    if the_num > largest_so_far :
        largest_so_far = the_num
    print(largest_so_far, the_num)

print('After', largest_so_far)
```

```
$ python largest.py
```

```
Before -1
```

```
9 9
```

```
41 41
```

```
41 12
```

```
41 3
```

```
74 74
```

```
74 15
```

```
After 74
```

We make a variable that contains the largest value we have seen so far. If the current number we are looking at is larger, it is the new largest value we have seen so far.



46

```

name = input('Enter file:')
handle = open(name, 'r')

counts = dict()
for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word,0) + 1

bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)

```

Sequential  
Repeated  
Conditional



```

name = input('Enter file:')
handle = open(name, 'r')

counts = dict()
for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word,0) + 1

bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)

```

Count words in a file

A word used to read data from a user

A sentence about updating one of the many counts

A paragraph about how to find the largest item in a list





## Exercise

- Write a program that reads a student's homework scores as input and computes the student's homework percentage.

```
This program computes your average homework grade.  
How many assignments were there? 3
```

```
Assignment 1  
Points earned? 12  
Points possible? 15
```

```
Assignment 2  
Points earned? 10  
Points possible? 20
```

```
Assignment 3  
Points earned? 4  
Points possible? 5
```

```
Your total score: 26 / 40 : 65 %
```



49

## while

- **while loop:** Executes 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
```



50

# try / except

```
astr = 'Hello Bob'  
try:  
    istr = int(astr)  
except:  
    istr = -1  
  
print('First', istr)
```



```
astr = '123'  
try:  
    istr = int(astr)  
except:  
    istr = -1  
  
print('Second', istr)
```



When the first conversion fails - it just drops into the except: clause and the program continues.

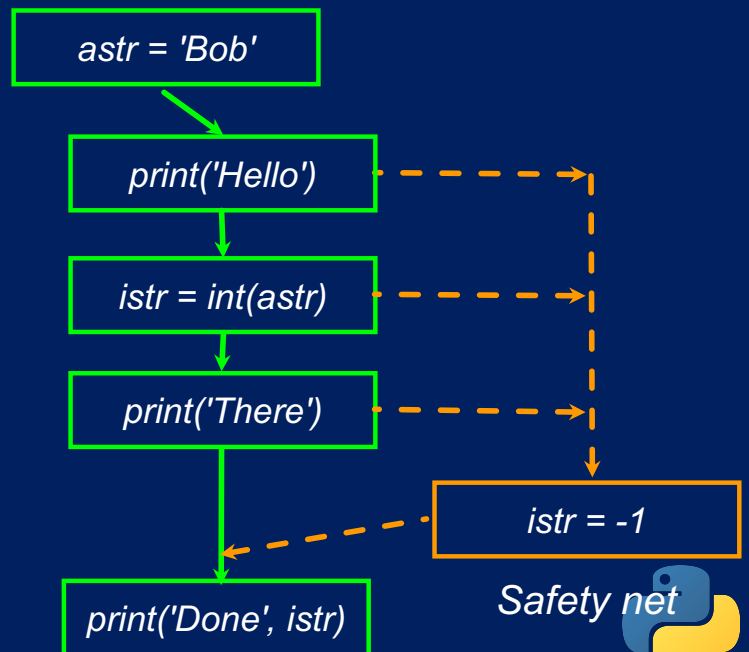
```
$ python tryexcept.py  
First -1  
Second 123
```

When the second conversion succeeds - it just skips the except: clause and the program continues.



# try / except

```
astr = 'Bob'  
try:  
    print('Hello')  
    istr = int(astr)  
    print('There')  
except:  
    istr = -1  
  
print('Done', istr)
```



Safety net



# Example

```
rawstr = input('Enter a number:')
try:
    ival = int(rawstr)
except:
    ival = -1

if ival > 0 :
    print('Nice work')
else:
    print('Not a number')
```

```
$ python3 trynum.py
Enter a number:42
Nice work
$ python3 trynum.py
Enter a number:forty-two
Not a number
$
```



53

# Random numbers

- `from random import *`
- `randint(min, max)`  
Produces a random value between **min** and **max** (inclusive)

## - Example:

```
coinflip = randint(1, 2)
if coinflip == 1:
    print "Heads"
else:
    print "Tails"
```



54

# Text Processing



## Strings

- **string:** A sequence of text characters in a program.
  - Strings start and end with quote " or apostrophe ' characters.

```
"hello"
```

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

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

- A string can represent special characters with a backslash.
  - \" quotation mark character
  - \t tab character
  - \\ backslash character

```
"Bob said, \"Hello!\" to Susan."
```



# Indexes

- Characters in a string are numbered with *indexes* :

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

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

- Accessing an individual character from a string:

**variable [ index ]**

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

Output:

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



57

# Looping Through Strings

- A definite loop using a for statement is much more elegant
- The iteration variable is completely taken care of by the for loop

```
fruit = 'banana'  
for letter in fruit :  
    print(letter)
```

```
index = 0  
while index < len(fruit) :  
    letter = fruit[index]  
    print(letter)  
    index = index + 1
```



58

# Slicing Strings

- Look at any continuous section of a string using a colon operator

M	o	n	t	y		P	y	t	h	o	n
0	1	2	3	4	5	6	7	8	9	10	11

```
>>> s = 'Monty Python'  
>>> print(s[:2])  
Mo  
>>> print(s[8:])  
thon  
>>> print(s[:])  
Monty Python
```

```
>>> print(s[0:4])  
Mont  
>>> print(s[6:7])  
P  
>>> print(s[6:20])  
Python
```



59

# String Concatenation

- When the + operator is applied to strings, it means "concatenation"

```
>>> a = 'Hello'  
>>> b = a + 'There'  
>>> print(b)  
HelloThere  
>>> c = a + ' ' + 'There'  
>>> print(c)  
Hello There  
>>>
```



60

# Strings Have Length

- The built-in function `len` gives us the length of a string

b	a	n	a	n	a
0	1	2	3	4	5

```
>>> fruit = 'banana'
>>> print(len(fruit))
6
```



61

# 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"
big_name = str.upper(name)
print big_name, "has", len(big_name), "characters"
```

**Output:**

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



62

# Data Types Conversion

- A string is a sequence of characters
- A string literal uses quotes 'Hello' or "Hello"
- For strings, + means "concatenate"
- When a string contains numbers, it is still a string
- We can convert numbers in a string into a number using int()

```
>>> str1 = "Hello"  
>>> str2 = 'there'  
>>> bob = str1 + str2  
>>> print(bob)  
Hellothere  
>>> str3 = '123'  
>>> str3 = str3 + 1  
Traceback (most recent call  
last): File "<stdin>", line 1,  
in <module>  
TypeError: cannot concatenate  
'str' and 'int' objects  
>>> x = int(str3) + 1  
>>> print(x)  
124  
>>>
```



63

# Text processing

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

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

Output:

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



64



# Strings and numbers

- `ord(text)` - Converts a string into a number.
  - Example: `ord("a")` is 97, `ord("b")` is 98, ...
  - Characters use standard mappings such as *ASCII* and *Unicode*.
- `chr(number)` - Converts a number into a string.
  - Example: `chr(99)` is "c"



65

## Exercise

- Write a program that "encrypts" a secret message by shifting the letters of the message by 1:
  - e.g. "Attack" when rotated by 1 becomes "buubdl"



66

# Functions



## Python Functions

- There are two kinds of functions in Python.
  - - Built-in functions that are provided as part of Python - `print()`, `input()`, `type()`, `float()`, `int()` ...
  - - Functions that we define ourselves and then use
- We treat the built-in function names as "new" reserved words (i.e., we avoid them as variable names)

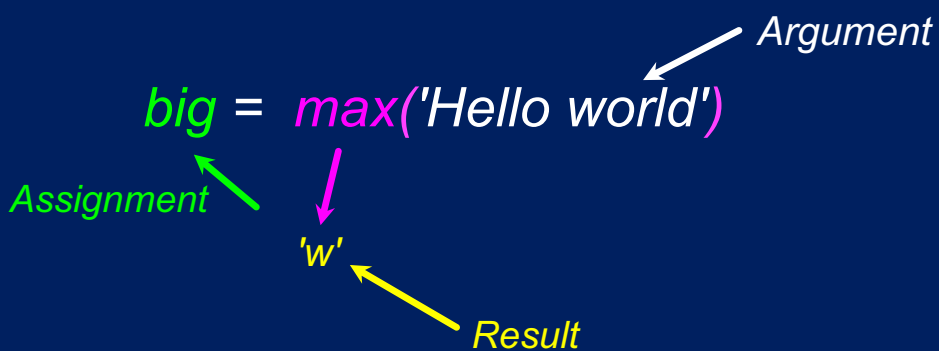


# Function Definition

- define a function using the `def` reserved word
- We call/invoke the function by using the function name, parentheses, and arguments in an expression



69



```
>>> big = max('Hello world')
>>> print(big)
w
>>> tiny = min('Hello world')
>>> print(tiny)

>>>
```



70

# Type Conversions

- When you put an integer and floating point in an expression, the integer is implicitly converted to a float
- You can control this with the built-in functions `int()` and `float()`

```
>>> print(float(99) / 100)
0.99
>>> i = 42
>>> type(i)
<class 'int'>
>>> f = float(i)
>>> print(f)
42.0
>>> type(f)
<class 'float'>
>>> print(1 + 2 * float(3) / 4 - 5)
-2.5
>>>
```



71

# Reading and Converting

- We prefer to read data in using strings and then parse and convert the data as we need
- This gives us more control over error situations and/or bad user input
- Input numbers must be converted from strings

```
>>> name = input('Enter:')
Enter:Chuck
>>> print(name)
Chuck
>>> apple = input('Enter:')
Enter:100
>>> x = apple - 10
Traceback (most recent call
last): File "<stdin>", line 1,
in <module>
TypeError: unsupported operand
type(s) for -: 'str' and 'int'
>>> x = int(apple) - 10
>>> print(x)
90
```



72

# Building our Own Functions

- We create a new function using the `def` keyword followed by optional parameters in parentheses
- We indent the body of the function
- This defines the function but does not execute the body of the function

```
def print_lyrics():  
    print("I'm a lumberjack, and I'm okay.")  
    print('I sleep all night and I work all day.')
```



73

# Arguments

- An argument is a value we pass into the function as its input when we call the function
- We use arguments so we can direct the function to do different kinds of work when we call it at different times
- We put the arguments in parentheses after the name of the function

```
big = max('Hello world')
```

Argument



74

# Parameters

- A parameter is a variable which we use in the function definition. It is a "handle" that allows the code in the function to access the arguments for a particular function invocation.

```
>>> def greet(lang):
...     if lang == 'es':
...         print('Hola')
...     elif lang == 'fr':
...         print('Bonjour')
...     else:
...         print('Hello')
...
>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
>>>
```



75

# Return Values

- Often a function will take its arguments, do some computation, and return a value to be used as the value of the function call in the calling expression. The return keyword is used for this.

```
def greet():
    return "Hello"

print(greet(), "Glenn")
print(greet(), "Sally")
```

Hello Glenn  
Hello Sally



76

# Reserved Words

- You cannot use *reserved words* as variable names / identifiers

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



77

# Files



# Opening and Reading a File

- A file handle open for read can be treated as a sequence of strings where each line in the file is a string in the sequence
- We can use the for statement to iterate through a sequence
- Remember - a sequence is an ordered set

```
xfile = open('mbox.txt')
for cheese in xfile:
    print(cheese)
```



79

# Counting Lines in a File

- Open a file read-only
- Use a for loop to read each line
- Count the lines and print out the number of lines

```
fhand = open('mbox.txt')
count = 0
for line in fhand:
    count = count + 1
print('Line Count:', count)
```

```
$ python open.py
Line Count: 132045
```



80



# Reading the \*Whole\* File

- We can read the whole file (newlines and all) into a single string

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```



81

# Bad File Names

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    quit()
```



82

# Lists



## A List is a Kind of Collection

- A collection allows us to put many values in a single “variable”
- A collection is nice because we can carry all many values around in one convenient package.

```
friends = [ 'Joseph', 'Glenn', 'Sally' ]
```

```
carryon = [ 'socks', 'shirt', 'perfume' ]
```



# Looking Inside Lists

- Just like strings, we can get at any single element in a list using an index specified in square brackets



```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]
>>> print(friends[1])
Glenn
>>>
```



85

# Lists are Mutable

- Strings are "immutable" - we cannot change the contents of a string - we must make a new string to make any change
- Lists are "mutable" - we can change an element of a list using the index operator

```
>>> fruit = 'Banana'
>>> fruit[0] = 'b'
Traceback
TypeError: 'str' object does not
support item assignment
>>> x = fruit.lower()
>>> print(x)
banana
>>> lotto = [2, 14, 26, 41, 63]
>>> print(lotto)
[2, 14, 26, 41, 63]
>>> lotto[2] = 28
>>> print(lotto)
[2, 14, 28, 41, 63]
```



86

## Example

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends :
    print('Happy New Year:', friend)

for i in range(len(friends)) :
    friend = friends[i]
    print('Happy New Year:', friend)
```

```
>>> friends = ['Joseph', 'Glenn', 'Sally']
>>> print(len(friends))
3
>>> print(range(len(friends)))
[0, 1, 2]
>>>
```

*Happy New Year: Joseph*  
*Happy New Year: Glenn*  
*Happy New Year: Sally*



87

## Concatenating Lists Using +

- We can create a new list by adding two existing lists together

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
>>> print(a)
[1, 2, 3]
```



88

## Lists Can Be Sliced Using :

```
>>> t = [9, 41, 12, 3, 74, 15]
>>> t[1:3]
[41, 12]
>>> t[:4]
[9, 41, 12, 3]
>>> t[3:]
[3, 74, 15]
>>> t[:]
[9, 41, 12, 3, 74, 15]
```

*Remember: Just like in strings, the second number is “up to but not including”*



89

## List Methods

```
>>> x = list()
>>> type(x)
<type 'list'>
>>> dir(x)
['append', 'clear', 'copy', 'count', 'extend',
 'index', 'insert', 'pop', 'remove', 'reverse',
 'sort']
>>>
```

<http://docs.python.org/tutorial/datastructures.html>



90

# Building a List from Scratch

- We can create an empty list and then add elements using the `append` method
- The list stays in order and new elements are added at the end of the list

```
>>> stuff = list()
>>> stuff.append('book')
>>> stuff.append(99)
>>> print(stuff)
['book', 99]
>>> stuff.append('cookie')
>>> print(stuff)
['book', 99, 'cookie']
```



91

# The Double Split Pattern

- Sometimes we split a line one way, and then grab one of the pieces of the line and split that piece again

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

```
words = line.split()
email = words[1]
print pieces[1]
```



92

# The Double Split Pattern

From **stephen.marquard@uct.ac.za** Sat Jan 5 09:14:16 2008

```
words = line.split()  
email = words[1]  
print pieces[1]
```

```
stephen.marquard@uct.ac.za
```



93

# The Double Split Pattern

From **stephen.marquard@uct.ac.za** Sat Jan 5 09:14:16 2008

```
words = line.split()  
email = words[1]  
pieces = email.split('@')  
print pieces[1]
```

```
stephen.marquard@uct.ac.za  
['stephen.marquard', 'uct.ac.za']
```



94

# The Double Split Pattern

From **stephen.marquard@uct.ac.za** Sat Jan 5 09:14:16 2008

```
words = line.split()
email = words[1]
pieces = email.split('@')
print(pieces[1])
```

stephen.marquard@uct.ac.za  
['stephen.marquard', 'uct.ac.za']  
'uct.ac.za'



95

## Dictionary





# Dictionaries

- Dictionaries are Python's most powerful data collection
- Dictionaries allow us to do fast database-like operations in Python
- Dictionaries have different names in different languages
  - - Associative Arrays - Perl / PHP
  - - Properties or Map or HashMap - Java
  - - Property Bag - C# / .Net



97

# Comparing Lists and Dictionaries

- Dictionaries are like lists except that they use keys instead of numbers to look up values

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print(lst)
[21, 183]
>>> lst[0] = 23
>>> print(lst)
[23, 183]
```

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print(ddd)
{'course': 182, 'age': 21}
>>> ddd['age'] = 23
>>> print(ddd)
{'course': 182, 'age': 23}
```



98

```

>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print(lst)
[21, 183]
>>> lst[0] = 23
>>> print(lst)
[23, 183]

```

*List*

Key	Value
[0]	21
[1]	183

*lst*

```

>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print(ddd)
{'course': 182, 'age': 21}
>>> ddd['age'] = 23
>>> print(ddd)
{'course': 182, 'age': 23}

```

*Dictionary*

Key	Value
['course']	182
['age']	21

*ddd*



99

## Counting with get()

- We can use get() and provide a default value of zero when the key is not yet in the dictionary - and then just add one

```

counts = dict()
names = ['csev', 'cwen', 'csev', 'zqian', 'cwen']
for name in names:
    counts[name] = counts.get(name, 0) + 1
print(counts)

```

Default

{'csev': 2, 'zqian': 1, 'cwen': 2}



100

# Counting Pattern

```
counts = dict()
print('Enter a line of text:')
line = input('')

words = line.split()

print('Words:', words)

print('Counting...')
for word in words:
    counts[word] = counts.get(word,0) + 1
print('Counts', counts)
```

The general pattern to count the words in a line of text is to *split* the line into words, then loop through the words and use a *dictionary* to track the count of each word independently.



101

# Retrieving Lists of Keys and Values

- You can get a list of keys, values, or items (both) from a dictionary

```
>>> jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> print(list(jjj))
['jan', 'chuck', 'fred']
>>> print(jjj.keys())
['jan', 'chuck', 'fred']
>>> print(jjj.values())
[100, 1, 42]
>>> print(jjj.items())
[('jan', 100), ('chuck', 1), ('fred', 42)]
>>>
```

What is a "tuple"? - coming soon...



102

# Credits

Charles R. Severance ([www.dr-chuck.com](http://www.dr-chuck.com)) of the University of Michigan School of Information

Marty Stepp of the University of Washington

