# Introduction to Computing

MCS1101B

Lecture 1: Python

# Preface

Download Python from
http://python.org/download/

IDLE Development Environment
Interactive interface with a *read-eval-print loop* (REPL)

Call python program via the python interpreter
**python fact.py**

All Python Documentations
https://docs.python.org/3/

Python Tutorial
https://docs.python.org/3/tutorial/index.html

fact.py

```
def fact(x):
    """Returns the factorial
    of its argument, assumed
    to be a posint"""
    if x == 0:
        return 1
    return x * fact(x - 1)

print ('N fact(N)')
print ("---------")

for n in range(10):
    print (n, fact(n))
```

# Python Scripts

- When you call a python program from the command line the interpreter evaluates each expression in the file
- Familiar mechanisms are used to provide command line arguments and/or redirect input and output
- Python also has mechanisms to allow a python program to act both as a script and as a module to be imported and used by another python program

# Example of a Script

```
x = 34 - 23                 # A comment.
y = "Hello"                 # Another comment.
z = 3.45
if z == 3.45 or y == "Hello":
    x = x + 1
    y = y + " World"        # String concatenation
print (x)
print (y)
```

# The Import System of Python

ex.py

```
def fact1(n):
    ans = 1
    for i in range(2,n):
        ans = ans * n
    return ans

def fact2(n):
    if n < 1:
        return 1
    else:
        return n * fact2(n - 1)
```

```
>>> import ex
>>> ex.fact1(6)
1296
>>> ex.fact2(200)
788657867364790503552363213932185O7…00000
   0L
>>> ex.fact1
<function fact1 at 0x902470>
>>> fact1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fact1' is not defined
```

# Enough to Understand the Code

**Indentation matters to code meaning**

- Block structure indicated by indentation

**First assignment to a variable creates it**

- Variable types don't need to be declared.
- Python figures out the variable types on its own.

**Assignment is = and comparison is ==**

**For numbers + - * / % are as expected**

- Special use of + for string concatenation and % for string formatting (as in C's printf)

**Logical operators are words (and, or, not) not symbols**

- The basic printing command is print

# Whitespace

**Whitespace is meaningful in Python**
*>> especially indentation and placement of newlines*

**Use a newline to end a line of code**

**Use \ when must go to next line prematurely**

**No braces { } to mark blocks of code, need to use consistent indentation instead**
*>> First line with less indentation is outside of the block*
*>> First line with more indentation starts a nested block*

**Colons start of a new block**
*>> function definitions, if clauses, etc.*

# Comments

# is used for single line comments
""" is used for multiline comments

Can include a *"documentation string"* as the first line of a new function or class you define

Development environments, debugger, and other tools use it: it's good style to include one

```
# defition for the factorial function

def fact(n):
  """fact(n) assumes n is a positive
    integer and returns facorial of
    n."""
  if n==1:
       return 1
  else:
       n*fact(n-1)
```

# Naming Rules

**Names are case sensitive and cannot start with a number**

They can contain **letters**, **numbers**, and **underscores**.

***Valid name examples:***

bob Bob _bob _2_bob_ bob_2

**There are some reserved words :**

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

**These shouldn't be used as names**

# Basic Datatypes

**Integer (default for numbers)**

```
z = 5 / 2
# Answer 2.5, normal division
z = 5 // 2
# Answer 2, integer division
```

**Float**

```
x = 3.456
```

There is to **Char** type in python

**String**

Can use `""` or `''` to specify

```
"abc"
'abc'
```

Unmatched " or ' can occur within the string:

```
"matt's"
'double quote symbol :"'
```

Use triple double-quotes for *multi-line strings* or *strings than contain both ' and "* inside of them: `"""a'b"c"""`

# Sequence Types

**Tuple**

('john', 32, [15,16,17], (10,20))

- A simple immutable ordered sequence of items
- Items can be of mixed types, including *collection types*

**String**

"John Smith"

- Immutable
- Conceptually very much like a tuple

**List**

[1, 2, 'john', ('up', 'down')]

- Mutable ordered sequence of items
- Can contain mixed types

# Assignment

You create a name the first time it appears on the left side of an assignment expression:

```
>>> x = 3
```

You can assign to multiple names at the same time

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

This makes it easy to swap values

```
>>> x, y = y, x
```

Assignments can be chained

```
>>> a = b = x = 2
```

Names in Python do not have an intrinsic type
data/objects have types

Python determines the type of the reference automatically based on what data is assigned to it

Binding a variable in Python means setting a name to hold some data value

Assignment always creates references, not copies

- For immutable types, it's does not matter
- For mutable types, it can create unexpected behaviors

# Defining Sequence Types

All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.

Key difference:

- Tuples and strings are immutable
- Lists are mutable

- Define tuples using parentheses and commas

```
>>> tu = (23, 'abc', 4.56, (2,3))
```

- Define lists are using square brackets and commas

```
>>> li = ["abc", 34, 4.34, (3,1,2)]
```

- Define strings using quotes (', ', or """)

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
                string that
uses                        triple
quotes."""
```

# Accessing Sequence Types 1

Access individual members of a tuple, list, or string using square bracket “array” notation

Note that all are 0-based…

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]     # Second item in the tuple.
'abc'
>>> li = ["abc", 34, 4.34, 23]
>>> li[1]      # Second item in the list.
34

>>> st = "Hello World"
>>> st[1]   # Second character in string.
'e'
```

# Accessing Sequence Types 2

Access individual members of a tuple, list, or string using square bracket “array” notation from the right-end using negative indexes

```
>>> x = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

**Positive index: count from the left, starting with 0**

```
>>> x[1]
‘abc’
```

**Negative index: count from right, starting with –1**

```
>>> x[-3]
4.56
```

# Slicing: return copy of a subset

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members.

Start copying at the first index, and stop copying before second.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

Negative indices count from end

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Omit first index to make copy starting from beginning of the container

```
>>> t[:2]
(23, 'abc')
```

Omit second index to make copy starting at first index and going to end

```
>>> t[2:]
(4.56, (2,3), 'def')
```

# Slicing: return a copy of a subset

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

[ : ] makes a copy of an entire sequence

```
>>> t[:]
(23, 'abc', 4.56, (2,3), 'def')
```

Note the difference between these two lines for mutable sequences

```
>>> x2 = x1 #Both refer to 1 ref,
        #changing one affects both
>>> x2 = x1[:]
     #Independent copies, two refs
```

Test your understanding:

```
>>> t[-1:-3] #??
>>> t[-1:2]  #??
>>> t[:-3]   #??
>>> t[1:4:2]   #??

>>> x1=[1,2,3]
>>> x2=x1   #x2=?
>>> x2[1] = 100
>>> x1        #??
>>> x3 = x1[:]   #x3=?
>>> x3[2] = 200
>>> x1        #??
```

# The ‘in’ Operator

Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

# The + Operator and The * Operator

The + operator produces a new tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]

>>> "Hello" + " " + "World"
'Hello World'
```

The * operator produces a new tuple, list, or string that “repeats” the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> “Hello” * 3
‘HelloHelloHello’
```

# Mutability : Tuples vs. Lists

```
>>> x = [‘abc’, 23, 4.34, 23]
>>> x[1] = 45
>>> x
[‘abc’, 45, 4.34, 23]
```

We can change lists in place.
The name x still points to the same memory reference when we’re done.

```
>>> t = (23, ‘abc’, 4.56, (2,3), ‘def’)
>>> t[2] = 3.14
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item
assignment
```

You can’t change a tuple.
You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

# Operations on Lists

```
>>> x = [1, 11, 3, 4, 5]

>>> x.append('a') # Note
the method syntax
>>> x
[1, 11, 3, 4, 5, 'a']

>>> x.insert(2, 'i')
>>>x
[1, 11, 'i', 3, 4, 5, 'a']
```

The extend method vs +
+ creates a fresh list with a new memory ref
extend operates on list x in place.

```
>>> x.extend([9, 8, 7])
>>> x
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

Potentially confusing:
extend takes a list as an argument.
append takes a singleton as an argument.

```
>>> x.append([10, 11, 12])
>>> x
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11,
12]]
```

# More on Tuples and Lists

The comma is the tuple creation operator, not parens

```
>>> 1,
(1,)
```

Python shows parens for clarity (best practice)

```
>>> (1,)
(1,)
```

Don't forget the comma!

```
>>> (1)
1
```

Trailing comma only required for singletons others

Empty tuples have a special syntactic form

```
>>> () #you can do the same using tuple()
()
```

Lists slower but more powerful than tuples

Lists can be modified, and they have lots of handy operations and methods

Tuples are immutable and have fewer features

To convert between tuples and lists use the **list()** and **tuple()** functions:

li = list(tu)

tu = tuple(li)

# Comparison of Python with C (in terms of speed)

```
import sys
NUMBER = int(sys.argv[1])
s = 0
for i in range(NUMBER):
    s += 1
```

time python python_loop.py **10000000**

real 0m1.044s
user 0m1.004s
sys 0m0.004s

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
 int NUMBER, i, s = 0;
 NUMBER = atoi(argv[1]);
 for (i = ; i < NUMBER; ++i)  s += 1;
 return 0;
}
```

**gcc c_loop.c -o c_loop**
time ./c_loop **450000000**

**real 0m1.032s**
**user 0m1.008s**
**sys 0m0.008s**

**gcc -O3 c_loop.c -o c_loop**
time ./c_loop **450000000**

**real 0m0.001s**
**user 0m0.000s**
**sys 0m0.000s**

# Next.

- You will learn about sets and dictionaries
- You will learn about python functions
- You will learn about file handling
- You will learn more about the import system