Administrivia

- Homework 2 is due **Wednesday**
- Homework 3 is due **Friday**
What is the value of x?

a) “125125”
b) 250
c) “1212”
d) 24
FUNCTIONS
Functions

• A small program we can run within Python
  – Saves us from having to rewrite code
  – Don’t reinvent the wheel!

• **ANALOGY**: Functions are *verbs* in Python.

• Also called a *subroutine* or *procedure*
Function calls

• When we want to execute a function, we call it or invoke it

• Use name of the function with parentheses
  – Example: print()

• Many functions are part of the Python language
  – We call them *built-in functions*
METHODS
Methods

• Like attributes, **functions** can be stored inside the type, too.

• Use **attribute operator** on the value.

  “STOP SHOUTING!” .lower()

  \((1+1j) .conjugate()\)

Value is treated like an argument.
String methods

"GATTACA".count("A")
"MVEMJSUN".find("J")
"ABACAB".replace("AB","G")
"HAM".strip()
"clint barton".title()
"wEiRd".swapcase()
s = "TACO TUESDAY"
x = s[0:s.find(" ")] 
   .lower()
x = x.title().swapcase()

What is the value of x?
a) "tACO"
b) "tuesdaY"
c) "Taco  

d) "TUESDAY"

WRITING FUNCTIONS
Defining functions

• We **define** a function by typing:
  1. the keyword **def**
  2. the name of our function
  3. a pair of parentheses
  4. a *block* of code
def greetings():
    print("Hola!")
    print("Bonjour!")
    print("Ni hao!")
    print("Hello!")
    print("Shalom!")
    print("Guten tag!")
    print(“Konnichiwa!”)
    print ("“As-salamu alaykum!”")
Block

• A section of code grouped together
• Begins with a colon :
• Contents of the block are *indented*
  – “Tabbed in”

def hello():
    print(“hello”)
Scope

• Variables declared *inside* a block are independent of variables *outside* the block.
• Variables inside a block *do not exist* outside the block.
• Blocks are their own little world!
• Blocks are *isolated* from the rest of our code.
a=5
def fun():
    a=3
    b=4
    a=a+b
    a=a+b
fun()
print(a)
a=5
def fun():
    a=3
    b=4
    a=a+b
    print(a)
Return

- Our function can `return` a value (output).
- We use the keyword `return`.

def three():
    return 3

- Return *immediately* exits the function.

def hello():
    return 0
    print(“hello”)
Parameters

• Our function can take *input* (arguments) as well.

• Parameters are variables declared in function header.

```python
def print_message(message):
    print(message)
```

• Multiple parameter are separated by commas.
def fun(a):
    return a+2

x=fun(2)*fun(3)

What is the value of x?
a) 9
b) 4
c) 16
d) None of the above.
def fun(m):
    return m.title().swapcase()

x = fun("abb") + fun("acab")

What is the value of x?
a) "AbbAcab"
b) "aBBaCAB"
c) "abbacab"
d) "ABBACAB"
def fun(a,b):
    c=((a+" ")*len(b)).title()
    x=fun("ab","caa")

What is the value of x?

a) "ab  ab  ab  ab"
b) "Ab  Ab  Ab"
c) "AB  AB  AB"
d) None of the above.
def fun(a,b):
    c=((a+(" "))*len(b)).title()
    return c

x=fun("ab","caa")

What is the value of x?
a) "ab  ab  ab"
b) "Ab  Ab  Ab"
c) "AB  AB  AB"
BOOLEAN TYPE
Booleans

- A type with only **two values**: 
  - **True** and **False**
- Used to represent **logic**
- We’ll use them to **make decisions**.
- Based on **Boolean algebra**
- Operators for Boolean type: 
  - and, or, not
### Logical operators

<table>
<thead>
<tr>
<th>and</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

- **True** when **BOTH** inputs are **True**

<table>
<thead>
<tr>
<th>or</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

- **True** if **ONE** input is **True**
### Logical operators

<table>
<thead>
<tr>
<th>not</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

Inverts the input
def fun():
    return True and False

x = fun() and not (True or False)

What is the value of x?
a) True
b) False
Comparison

- Operators that produce Boolean output
  
  <  less than
  <= less than or equal <=
  >  greater than
  >= greater than or equal
  == equal
  != not equal
a = 5
b = 3

\[ x = (a < 5) \text{ and } ((b \leq 5) \text{ or } (a \neq b)) \]

What is the value of x?
a) True
b) False
a = "HAWKEYE"
b = "IRON MAN"

x = a < b and a[1] != b[-2]

What is the value of x?
a) True
b) False
def fun(a, b):
    return a < b

a = 3
b = 4

x = fun(b, a)

What is the value of x?
a) True
b) False
a=1
def fun(a,b):
    return a+b
a=fun(a,a)+a

What is the value of a?
a) 2
b) 3
c) 4
d) None of the above.
a=1
def fun(c,b):
    return c+b
a=fun(a,a)+a

What is the value of a?
a) 2
b) 3
c) 4
d) None of the above.
CONDITIONAL EXECUTION
Conditional Execution

• Make decisions in our program
• Change program behavior
  – Based on a Boolean value
• Change the *control flow*
If statement

• We create an *if statement* by typing:
  1. the keyword *if*
  2. a Boolean expression
  3. a *block* of code
print("Welcome to my program.")
answer=input("Are you nice?")
if answer=="Yes":
    print("Hello, friend!")
x=10
if ((x/2)<5) or ((x%3)==1):
    x=x+2
if (x!=10) or ((x**2)<=144):
    x=x*2

What is the value of x?
a) 10
b) 12
c) 20
d) 24
Alternative Execution

- Make decisions in our program
- Change program behavior
- Change the *control flow*
- Execute one block OR another block
If... else statement

- We create an *if... else statement* with:
  1. the keyword *if*
  2. a Boolean expression
  3. a *block* of code
  4. the keyword *else*
  5. another *block* of code
print("Welcome to my program.")
answer=input("Are you nice?")
if answer=="Yes":
    print("Hello, friend!")
else:
    print("HEY! BE NICE!")
Alternative Execution

• Make decisions in our program
• Change program behavior
• Change the *control flow*
• Execute one block *or* another
Alternative Execution

True

if block

False

else block
If... else statement

- We create an *if... else statement* with:
  1. the keyword *if*
  2. a Boolean expression
  3. a *block* of code
  4. the keyword *else*
  5. another *block* of code
print("Welcome to my program.")
answer=input("Are you nice?")
if answer==="Yes":
    print("Hello, friend!")
else:
    print("HEY! BE NICE!")
def absolute(x):
    if x>=0:
        return x
    else:
        return -x
BOOLEANS AND STRINGS
Boolean string methods

• isdigit() - “Is the string all digits?”
• isalpha() - “Is the string all letters?”
• islower() - “Is the string all lower case?”
• isupper() - “Is the string all upper case?”
answer=input("Are you nice?")
if not answer.isalpha():
    print("I don’t understand.")
else:
    print("I think I understand.")
Sequence operators

• in “Is this string inside the other?”
• not in “Is this string NOT inside the other?”
def fun(s):
    return s.isalpha() and "s" in s

x = fun("sam") and fun("AS")

What is the value of x?

a) True
b) False
def fun(x):
    if x and x:
        return not x
    else:
        return x or x

x = fun(True) or fun(False)

What is the value of x?

a) True
b) False
def fun(a,b):
    if len(a)+len(b)>5:
        return (a+b)[0:5]
    else:
        return (b+a)+str(len(a))

x=fun("abc","def")+fun("gh","ij")

What is the value of x?
a) “abcdefijgh4 
b) “defabcghij4”
c) “abcdeijgh4”
d) None of the above.
def fun(x):
    if x<100 or ???:
        return x+1
    else:
        return x

Increment x if the 100’s place is 0:
a)x.string(3)==‘0’
b)str(x)[-3]==‘0’
c)((x/100)%10)==0
d)None of the above.
Nesting

• Sometimes, we need to make *more than one decision*

• We can *nest* blocks
  – One block inside the other
  – We’ve already been nesting conditionals and functions
answer = input("Hello!")
if not answer.isalpha():
    print("I don't understand.")
else:
    if answer.isupper():
        print("Don't shout!")
    else:
        print("Thanks!")
“I don’t understand.”

“Don’t shout!”

“Thanks!”

not alphabetical?

True

False

all upper case?

True

False
x is positive?

- True
  - x is even?
    - True
      - x
    - False
      - x + 1
- False
  - x is even?
    - True
      - -x
    - False
      - (-x) + 1
def evenpos(x):
    if x>=0:
        if (x%2)==0:
            return x
        else:
            return x+1
    else:
        if (x%2)==0:
            return -x
        else:
            return (-x)+1
Multi-way branch

- Sometimes, we want to choose between multiple choices
- Example: day of week => string
if day==1:
    print(“Sunday”)
else:
    if day==2:
        print(“Monday”)
    else:
        if day==3:
            print(“Tuesday”)
        else:
            if day==4:
                print(“Wednesday”)
            else:
                if day==5:
                    print(“Thursday”)
                else:
                    if day==6:
                        print(“Friday”)
                    else:
                        if day==7:
                            print(“Saturday”)

This sucks!
elif

- Shorthand for `else if`
- We don’t need to indent again!
if day==1:
    print("Sunday")
elif day==2:
    print("Monday")
elif day==3:
    print("Tuesday")
elif day==4:
    print("Wednesday")
elif day==5:
    print("Thursday")
elif day==6:
    print("Friday")
elif day==7:
    print("Saturday")
else:
    print("That is not a valid day.")
s = "ABcd"

if not s[0:2].isupper():
    if s[0] == s[2]:
        print(s[0])
    else:
        print(s[1])
else:
    if s[1] != s[2]:
        print(s[-1])
    else:
        print(s[-2])
s="abcd"
if not s.isalpha():
    print(s[0])
elif s.isupper():
    print(s[-1])
elif "ab" in s:
    print(s[-2])
else:
    print(s[1])
Exercise

• Validate password
  – At least 8 characters long
  – Upper and lower case characters
  – At least one non-alphabetic character
  – First three symbols must be distinct

• validate_password("ABC") → False
• validate_password("AA9aaaaa") → True
def validate_password(password):
    if not len(password) >= 8:
        return False
    elif password.isupper():
        return False
    elif password.islower():
        return False
    elif password.isalpha():
        return False
    elif password[0] == password[1]:
        return False
    elif password[1] == password[2]:
        return False
    elif password[0] == password[2]:
        return False
    else:
        return True
QUICK ASIDE
Shorthand

• `a+=b` is shorthand for `a=a+b`
• `a-=b` is shorthand for `a=a-b`
• `a*=b` is shorthand for `a=a*b`
• `a/=b` is shorthand for `a=a/b`
LOOPING
While loop

• Allows for *repeated execution* of code
• Execute a block over and over as long as a Boolean condition is True
• *Stop executing* if Boolean condition is False
While loop

• We create an **while loop** by typing:
  1. the keyword **while**
  2. a Boolean expression
  3. a **block** of code
x=3
while x>0:
    print("Hello")
    x-=1

How many times is “Hello” printed?

a) 0
b) 1
c) 2
d) 3
e) 4
Exercise

• Password creation:
  – Call validate password
  – Repeat until user inputs a valid password.
Solution

```
pwd=input("Enter a password: ")
while not validate_password(pwd):
    pwd=input("INVALID! Reenter: ")

print("Your password is valid")
```
Infinite loop

while True:
    print "Hello"

• **ALWAYS**: Statements *inside* the loop *must* change the loop condition!

• CTRL-C will stop the loop
Accumulator pattern

• Common and useful pattern to design programs
• *Accumulator* variable keeps track of result
  – Updated in each loop iteration
\[ i = 0 \]

\[ \text{sum} = 0 \]

while \( i \leq 4 \):
    \[ \text{sum} += i \]
    \[ i += 1 \]

a) 6
b) 10
c) 15
d) None of the other answers.
i=0
sum=0
while i<7:
    if (i%2)==1:
        sum+=i
        i+=1

a) 9
b) 12
c) 16
d) 21
Exercise

- Write a function to sum all of the digits in a number

\[
\text{sum}(12145) \rightarrow 1+2+1+4+5 \rightarrow 13
\]
def sum_digits(n):
    s = str(n)
    i = 0
    result = 0
    while i < len(s):
        result = result + int(s[i])
        i = i + 1
    return result
s="ABCDEFGH"
t=""
i=0
while i<8:
    t=t+s[i+1]
    i+=2

What is the final value of t?

a) "ACEG"
b) "BDFH"
c) "ABCDEF"
d) "ABEF"
s="0123456789"
t=""
i=0

while i<5:
    if (i%2)==1:
        t=t+s[i-1]
    else:
        t=t+s[i+1]

i=i+1

What is the final value of t?

a) "92143"
b) "103254"
c) "10325"
d) "921436"
FOR LOOPS
For loop

• Loop construct to make our lives easier
• Used to iterate over *iterable* types
  – Example: strings (more to come)
• Step through a sequence “one at a time”
For loop

• We create an *for loop* by typing:
  1. the keyword *for*
  2. a loop variable (just a variable)
  3. the keyword *in*
  4. an iterable
  5. a *block* of code
Example

my_string="abcdefg"
for letter in my_string:
    print(letter)
def sum_digits(n):
    result = 0
    for letter in str(n):
        result += int(letter)
    return result
s="abcdefg"
t=""
for c in s:
    t=c+t
What is the value of t?
a) “abcdefg”
b) “gfedcba”
c) “a”
d) “g”
s = "abcdefg"

t = ""

for c in s:
    t = c + t

What is the value of t?

a) "abcdefg"
b) "gfedcba"
c) "a"
d) "g"
s="Run The Jewels"
t=""
for c in s:
    if c.isupper():
        t=t+c.lower()

What is the value of t?

a) “RTJ”
b) “un he ewels”
c) “”
d) “rtj”
FILE INPUT
Files

- Iterable type
- Created with built-in function `open()`
- 1 argument: file name as a string (for now)
- Each item in the iterable is a string representing one line in the file

```python
for line in open("words.txt")
    print(line)
```
Example

total=0
for line in open("numbers.txt"):
    total+=int(line)
print(total)
Example

for w in open("words.txt"):  
vowels=0  
for c in w.lower():  
    if c in 'aeiou':  
        vowels+=1  
print(w.strip()+" %i" % vowels)
LISTS
Lists

• Represents an ordered collection of *items* or *elements*
  – Another *iterable* type

• Our first *container* type
  – Contains other values of *any type*
  – **NOTE**: elements don’t have to be the same type
Lists

• We create an *list* by typing:
  1. an open square bracket `[`
  2. items of the list, separated by commas
  3. a closing square bracket `]`
Similarity to Strings

```python
x=[10,3.14,“Ride”]
print(x[1])
print(x[1:3])
print(len(x))
for i in x:
    print(i)
```
Dissimilarity to Strings

• Strings are **immutable** (we can’t change the *contents* without **creating a new string**)

s = “Puraty Ring”

s[3] = “i” ← NOT ALLOWED

s = s[0:3] + “i” + s[4:]
Immutable Assignment

\[ x = 3.14 \]
\[ y = x \]

Memory

```
x -> 3.14
y -> 3.14
```
Immutable Assignment

s = "Purity Ring"

Memory

$t = s$

\[ \text{Memory} \]

\[ \text{s} \rightarrow \text{"Purity Ring"} \]

\[ \text{t} \rightarrow \text{"Purity Ring"} \]
Dissimilarity to Strings

- Lists are **mutable** (we *can* change the contents of a list)

```
x=[4,1,2,3]
x[3]=-2  # item assignment
x.append(5)
del x[1]
x.sort()
```
Mutable Assignment

\[
a = [1, 2, 3, 4] \\
b = a
\]

Memory
Aliasing

• One memory location has two names.
• Only *mutable* types can be aliased.
• Aliasing causes mutable types to behave *very* differently.
Implications of Aliasing

\[ a = [1, 2, 3, 4] \]
\[ b = a \]
\[ b[-1] = 2 \]
x=[3,2,1]
y=x
y.sort()
x.append(0)

What is the final value of x?
a) [3,2,1]
b) [1,2,3]
c) [1,2,3,0]
d) [0,1,2,3]
DANGER!! DANGER!!

- The `sort` and `append` methods modify the list *itself*
- This means they **RETURN NULL**

```python
x = [1, 2, 3]
print(len(x))
x = x.append(5)
```
```python
y=[3,2,1]
x=y.append(5)
y[-1]=3

What is the final value of x?
a) [3,2,1,3]  
b) [3,2,1,5]  
c) [3,2,1,5,3]  
d) None
```
Range

- The `range function` returns an immutable iterator containing integers
- Can be cast to a list
- Two arguments:
  - the starting value our range
  - the ending (not included!) value in our range

```python
x=list(range(2,5))
```
Example

total=0
for x in range(0,1000):
    total=total+x
print(total)
Example

total=0
for x in range(0,1000):
    prime=True
    for y in range(2,x):
        if (x%y)==0:
            prime=False
    print(x)