COSC 2306
Data Programming

Python Basics

What’s the output?

```
x = -10
if x < 0:
    print("The negative number ", x, " is not valid here.")

print("This is always printed")
```

The negative number -10 is not valid here.
This is always printed

What's the output?

```
x = 10
if x < 0:
    print("The negative number ", x, " is not valid here.")

    print("This is always printed")
```

Python uses indentations to control logics

What's the output?

```
x = -10
if x < 0:
      print("The negative number ", x, " is not
      valid here.")
else: print(x, " is a positive number")
else: print(x," is 0")
```

- Syntax error

What's the output?

```
x = -10
if x < 0:
    print(x, " is a negative number.")
elif x >0:
    print(x, " is a positive number")
else:
    print(x, " is 0")
```

```
-10 is a negative number.
```

What's the output?

```
x = -10
if x < 0:
        print(x, " is a negative number")
else:
        if x > 0: print(x, " is a positive number")
        else: print(x," is 0")
```

```
-10 is a negative number
```

-You can have an if/else statement under an else statement

Loops/Iteration

- Loops allow us to specify a set of statements that need to be repeated several times

- We use computers to automate tasks - they are really good at doing the same thing over and over again

- Python provides language features to support iterations in our programs:
  - **`for loop`**
  - **`while loop`**

for and while

To repeat a block of code multiple times, we can use **loops (while or for)**:

- A while loop will execute until some condition is true

Don't forget to initialize and update controlling variable

```
while x < 10:
      #do something
```

```
for x in range(10):
      #do something
```

- A for loop is better for known iterations and counting

for Loop

- The `for` statement allows us to iterate through a sequence of values

- ```
  for <var> in <sequence>:
      <body>
  ```

- The loop index variable `var` takes on each successive value in the sequence, and the statements in the body of the loop are executed once for each value
  ```

Very Simple Example

- `for fruit in ["Apples", "Bananas", "Grapes", "Oranges"]:`
- `    print ("We have " + fruit)`

- Output:

We have Apples
We have Bananas
We have Grapes
We have Oranges

Simple Counting Program

- `sum = 0`
- `for number in [0, 1, 2, 3, 4, 5]:`
```
      sum = sum + number
print("The total is", sum)
```

- `sum = 0`
- `for number in range(6):`
```
      sum = sum + number
print("The total is", sum)
```

Range

- **Range**(n)
  - n is an integer
  - Iterates through 0, 1, …… n-1
- **Range**(start, end, step)
  - Iterate from *start* to *end-1* with step length *step*
  - Range(1, 7, 2): 1, 3, 5

- Range() works like a list

For iteration

- Iterator variable reset at the beginning of each iteration

```
for number in range(5):
    number = number + 2
    print(number)
```

For iteration

- Iterator variable reset at the beginning of each iteration

```
for number in range(5):
    number = number + 2
    print(number)
```

```
2
3
4
5
6
```

Limitation of for Loops

- The `for` loop is a *definite* loop, meaning that the number of iterations is determined when the loop `starts`

- *But the number may not always be known!*

Example Problems

- A program that asks a **user** to enter a sequence of positive integers, with a "-1" entered to indicate the end of the sequence (then the program computes and prints the sum of the numbers)
  - This is a common pattern where the end of iterations is indicated by a special value
- A program that counts vowels in an input sentence?
- A program that totals your grocery bill?

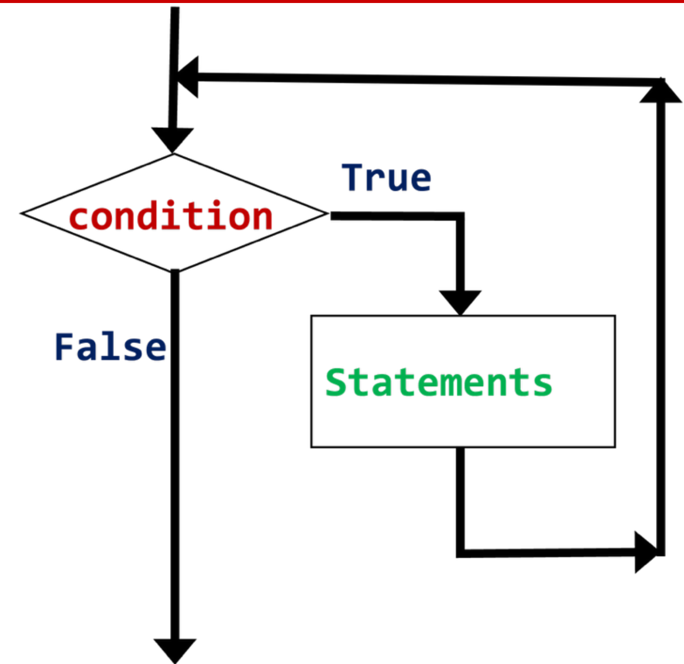
Example Problems

- A program that accepts donations and stops when the total reaches $1000?
  - In this pattern, condition to stop iterating is computed dynamically
- The program randomly picks a number between 1 and 100. The user guesses repeatedly, and program say "hot", "cold, or "bingo"
- A program that computes smallest P such that 3^P is greater than 1 million

while Loop

Syntax:

```
while <condition>:
            <statements>
```



Here's the flow of execution:

1. Evaluate the condition, yielding True or False
2. If condition is True, then execute the body of loop and go back to step 1
3. If the condition is False, then exit the while statement and continue execution at the next statement

Example of a while Loop

- Sum the first 4 integers

```
sum = 0

for k in range(5):
    sum = sum + k
print(sum)
```

```
k = 0
sum = 0

while k < 5:
    k = k + 1
    sum = sum + k
print(sum)
```

Countdown

```
for i in [5,4,3,2,1]:
    print(i)
print("Blast off")
```

```
n=5
while n > 0:
    print(n)
    n = n-1
print("Blast off")
```

Sum a list of numbers

Write a program that asks a user to enter a sequence of positive integers, with a "*-1*" entered to indicate the end of the sequence. Then the program computes and prints the sum of the numbers?

```
sum = 0
newnumber = int(input("Enter next number "))
while newnumber > 0:
    sum = sum + newnumber
    newnumber = int(input("Enter next number "))
print("The total is ", sum)
```

Example Problems

A program that accepts donations and stops when the total reaches or exceeds $1000?

```
max = 1000.0
current_total =0.0
while current_total < max:
    donation = float(input("Enter amount "))
    current_total = current_total + donation
print ("Mission accomplished! Total raised $ ",
current_total)
```

Continue and Break in Loops

- Sometimes we need to skip the current round of loop, or terminate the whole loop.
  - **continue**: skip the current round, directly go to the next.
  - **break**: Terminate the loop.

```
for k in range(5):
    if k == 3:
        continue
    print(k)
```

```
for k in range(5):
    if k == 3:
        break
    print(k)
```

Continue and Break in Loops

- Sometimes we need to skip the current round of loop, or terminate the whole loop.
  - **continue**: skip the current round, directly go to the next.
  - **break**: Terminate the loop.

```
for k in range(5):
    if k == 3:
        continue
    print(k)

[0,1,2,4]
```

```
for k in range(5):
    if k == 3:
        break
    print(k)

[0,1,2]
```

Continue and Break in Loops

- Continue and break only works for one-layer of loop

```
for i in range(2):
    for k in range(4):
        if k == 2:
            break
        print(i,k)
```

Continue and Break in Loops

- Continue and break only works for one-layer of loop

```
for i in range(2):
    for k in range(4):
        if k == 2:
            break
        print(i,k)
```

```
0 0
0 1
1 0
1 1
```

The i-loop is not interrupted.

while Loop vs. for Loop

- A for loop is less effort on the programmer's side
- If we know the number of iterations needed, then a for loop is typically better
- When we don't know beforehand how many iterations, then while loop is better
- Any for loop problem can be solved with a while loop but not the other way round

Functions: what and why

- Functions are blocks of code grouped together to perform a task
- Advantages of functions:
  - Makes the code more readable
  - They are reusable
  - Black boxes

```
num = int(input())
result = num * num
```

```
import math

num = int(input())
result = math.pow(num,2)
```

Built-in functions

- Many functions are already available in existing Python -- **modules (use import)**
- Example: the random module
  - random.random() → returns a floating point number in the range [0.0, 1.0)
  - random.randrange(x,y) → returns an integer number in the range [x, y)

Questions: how can you generate a floating point random number between 0 and 100? And between 10 and 100?

```
import random
random.random() * 100
random.random() * 90 + 10
```

User defined functions

It's very easy to create new functions in Python

```
def newFunction( arguments ):
        action 1
        action 2
        …
```

Example: let's create a function to greet the user

```
def greeting(name):
        print ("Hello! ", name, "!", sep="")
```

Fruitful or not?

- The greeting function is unfruitful → it does not return a value

- A fruitful function includes a return statement:

```
def square(x):
    y = x * x
    return y

toSquare = 10
result = square(toSquare)
print("The result of", toSquare, "squared is", result)

> The result of 10 squared is 100
```

The `return` statement

- Ends the function
  - Not always at the end of the function

```
def isItEven(x):
    if x % 2 == 0:
        return True
    else:
        return False

print(isItEven(10))

>True
```

Fruitful or not?

```
def square(x):
    y = x * x
    print(y)

toSquare = 10
result = square(toSquare)
print("The result of", toSquare, "squared is", result)
```

```
>100
>The result of 10 squared is None
```

*This is not a fruitful function since “None” was return

Local variables

- Variables created inside a function can not be used outside → local variables
- Arguments are also local variables
- Global variables can be used, but it is not good practice

```
def badsquare(x):
       y = x ** power
       return y

power = 2
result = badsquare(10)
print(result)

>100
```

What happens if I try to change power in the function?
E.g., power = 3

>1000

Passing argument

```
def reassign(list):
  list = [0, 1]

list = [0]
reassign(list)
print (list)


>[0]
```

```
def append(list):
  list.append(1)

list = [0]
append(list)
print (list)


>[0, 1]
```

Passing argument

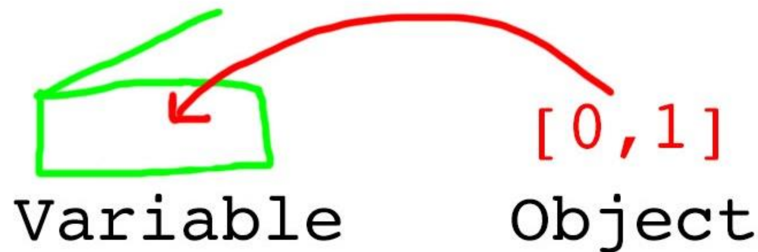
- Commonly, parameters are passed to a function by reference or by value
- Java is always pass by value, C++ has both
- … but not in Python
- Reference: https://robertheaton.com/2014/02/09/pythons-pass-by-object-reference-as-explained-by-philip-k-dick/

Pass-by-object-reference

The variable is not the object

a = []

[] is the empty list object
a is a variable that points to the empty list object
a itself is not the empty list object



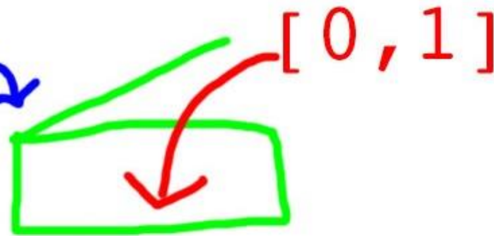
Pass-by-reference

Pass-by-reference

the box (the variable) is passed directly into the function, and its contents (the object represented by the variable) implicitly come with it, the **same location in memory**

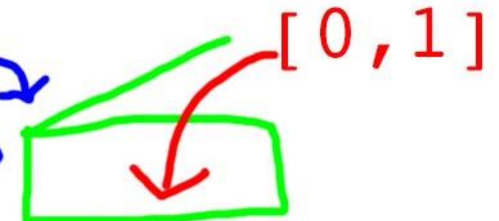
```
def reassign(list):
  list = [0,1]

list = [0]
reassign(list)
print list
```




```
def append(list):
  list.append(1)

list = [0]
append(list)
print list
```

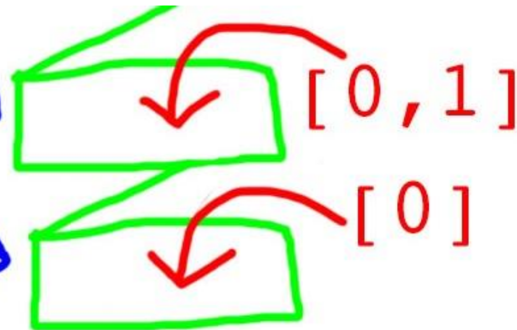


Pass-by-value

Pass-by-value the function receives a **copy** of the argument objects, which is stored in a **new location in memory**

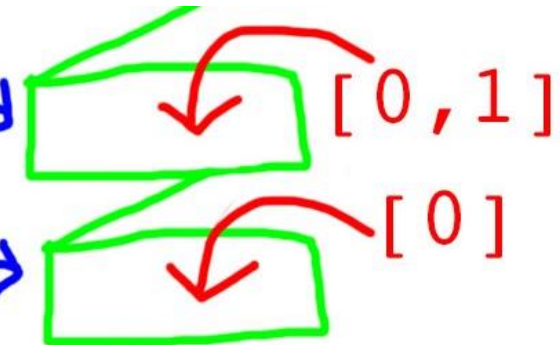
```
def reassign(list):
  list = [0,1]

list = [0]
reassign(list)
print list
```




```
def append(list):
  list.append(1)

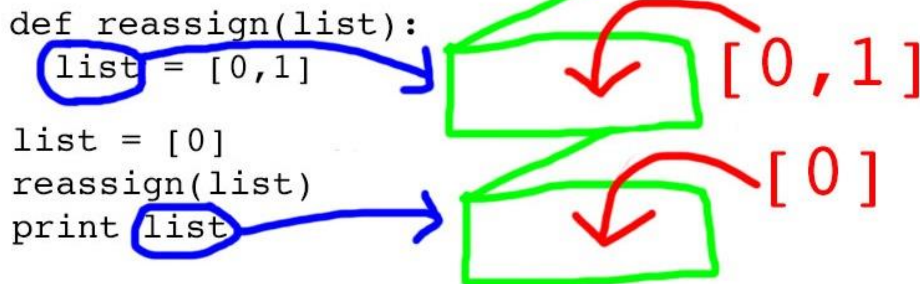
list = [0]
append(list)
print list
```



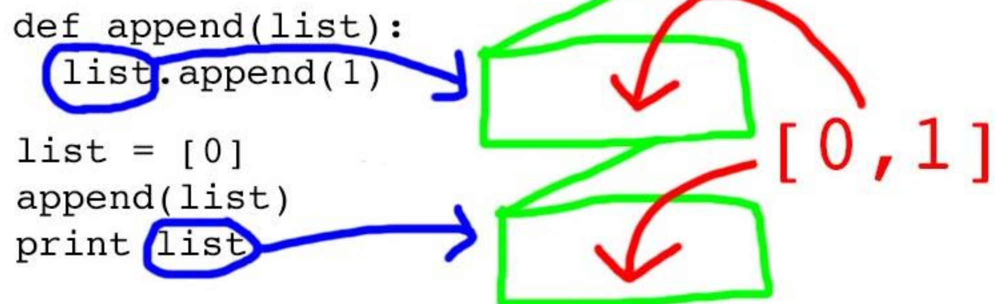
Pass-by-object-reference

Pass-by-object-reference

- the function receives a **reference** to the same object in memory as used by the caller (similar as reference)
- it does not receive the box that the caller is storing this object in
- the function provides its own box and creates a **new variable** for itself (similar as pass-by-value)



The caller doesn't care if you ***reassign*** the function's box



Both the function and the caller refer to the same object in memory, so *append* adds an extra item to the list, we see this in the caller too

Reference

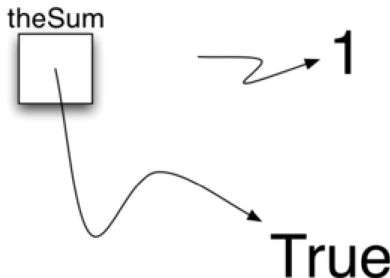
- Variable: hold reference to a piece of data (not data itself)

- Assignment statement: associate a name (variable) with a piece of data (object)

```
theSum = 0
print(theSum)
>> 0
```

```
theSum = theSum + 1
print(theSum)
>> 1

theSum = True
print(theSum)
>> True
```



- **reassign may change data type**
- **the same variable can refer to many different types of data**

Mutable vs Immutable

- Every piece of data in Python is an object
- Every variable reference to an object instance
- When an object is initiated, it is assigned a **unique object id**
- Object **type** is defined at runtime and once set can never change
- Object **state** can be changed if it is mutable
- Sum up: a mutable object can be changed after it is created, and an immutable object can't

Why?

Mutable: flexible, less recreation; immutable: data consistency, efficiency