ICT 133
Structured Programming

Seminar 4

Topics

- File Loop
- Scalar vs Sequence types
- Immutable vs mutable types
- Scope of Function Parameters
- Functions with immutable and mutable parameters
- Top down design

Files: Multi-line Strings

- A *file* is a sequence of data stored in secondary memory
- A file usually contains more than one lines.
  - We focus on text file.
  - Python uses the standard newline character (`\n`) to mark line breaks.
  - For example:

```
Hello
World

Goodbye 32
```

When stored in a file:

```
Hello\nWorld\n\nGoodbye 32\n
```

File Processing

- Opening a file

```
    <filevar> = open(<name>, <mode>)
```

Examples:

```
infile = open("numbers.dat", "r")
outfile = open("mydata.out", "w")
```

- Closing the file completes any outstanding operations and bookkeeping for the file

```
    <filevar>.close()
```

Examples:

```
infile.close()
outfile.close()
```

File Loops - reading

```
def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName,'r')
    sum, count = 0.0, 0
    for line in infile: # lines separated by \n
        sum = sum + float(line)
        count = count + 1
    print("\nThe average of the numbers is", sum/count)
    infile.close()
```

```
File:
1
3
4
```

Writing to File

```
def main():
    fileName = input( "What file are the numbers in? ")
    infile = open(fileName,'r')
    outfile = open(fileName + '.out', 'w')
    sum, count = 0.0, 0
    for line in infile: # lines separated by \n
        sum = sum + float(line)
        count = count + 1
    print( "|nThe average of the numbers is", sum/count, file =
outfile)
    infile.close(); outfile.close()
```

Nested File Loops

```
def main():
    fileName = input("What file are the numbers in? ")
    infile = open(fileName,'r')
    sum, count = 0.0, 0
    for line in infile:
        for xStr in line.split(","):
            sum = sum + float(xStr)
            count = count + 1
    print("\nThe average of the numbers is", sum/count)
    infile.close()
```

File:
1,2
3
4,5,6

File Methods

`<file>.read()`	Returns the unread content as a single string
`<file>.readline()`	Returns the next line of the file.
`<file>.readlines()`	Returns a sequence (a list) of unread lines in the file.
`<file>.write(str)`	writes string to the file, and return the number of characters.
`<file>.close()`	Closes file and release resources

Python Data Value

- Every data in Python is an object.
- An object has
  - content (the value), e.g., 3
  - type (the data type of the value) e.g., int
  - id or an identity (the address where the value is stored in memory) e.g., `493790368`

Scalar Data Type

- Single value
  - `int` e.g., 3, -4, 0
  - `float` e.g., 3.0, -0.2523

```
>>> type(3)
<class 'int'>
>>> type(3.0)
<class 'float'>
```

```
>>> myInt = 3
>>> type(myInt)
<class 'int'>
>>> id(myInt)
493790368
>>> id(3)
493790368
```

Sequence Data Type

- Values (or **elements**) are ordered in a collection e.g., `str`“Hello“

```
>>>greet = "Hello"
>>>greet
'Hello`
>>>id(greet)
108421048
>>>id("Hello")
108421048
>>>type(greet)
<class 'str'>
>>>type("Hello")
<class 'str'>
```

```
>>>greet[3]
'l'
>>>type(greet[3])
<class 'str'>
>>>id(greet[3])
33562496
```

Other Sequence Data Type

- **`list`**
  - Elements are values of any data type, enclosed within square brackets
    e.g., `[1, 2, 'Ann', 3.3]`

- **`tuple`**
  - sequence of values of any data type, enclosed within round brackets
    e.g., `(1, 2, 'Ann', 3.3)`

Accessing Elements

- Individual elements through *indexing*.
  `[1, 2, 'Ann', 3.3][2]` evaluates to `'Ann'`

- A contiguous sequence of elements through *slicing*.
  `[1, 2, 'Ann', 3.3][2:]` evaluates to
  `['Ann', 3.3]`

- Iteration through elements

```
for elem in [1, 2, 'Ann', 3.3]:
     print (elem, end=" ")
```

Output:

```
1 2 Ann 3.3
```

Combining Elements

- Similar to str
  - *Concatenation* “glues” two sequences together (+)
  - *Repetition* builds up a string by multiple concatenations of a string with itself (*)

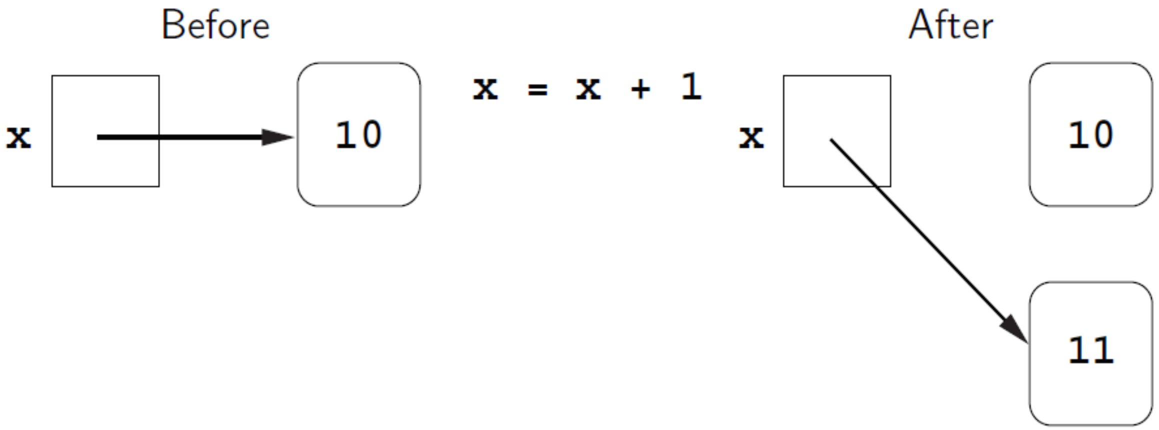
```
>>> t1 = (1, 2)
>>> t2 = (3,)

>>> t1 + t2
(1, 2, 3)

>>> t2*5
(3, 3, 3, 3, 3)
```

Immutable Data Types

- `int`, `float`, `str` and `tuple`
  - Values cannot be changed without changing the identities



```
>>> id(10)
493790592
>>> id(11)
493790624
```

```
>>> x = 10
>>> id(x)
493790592
```

```
>>> x = x + 1
>>> x
11
```

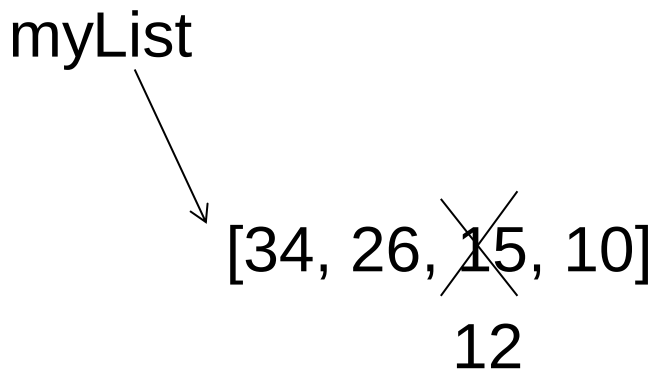
```
>>> id(x)
493790624
```

Mutable Data Types

- `list`
  - Values can be changed without changing the identities

```
myList = [34, 26, 15, 10]

myList[2] = 12
```



myList

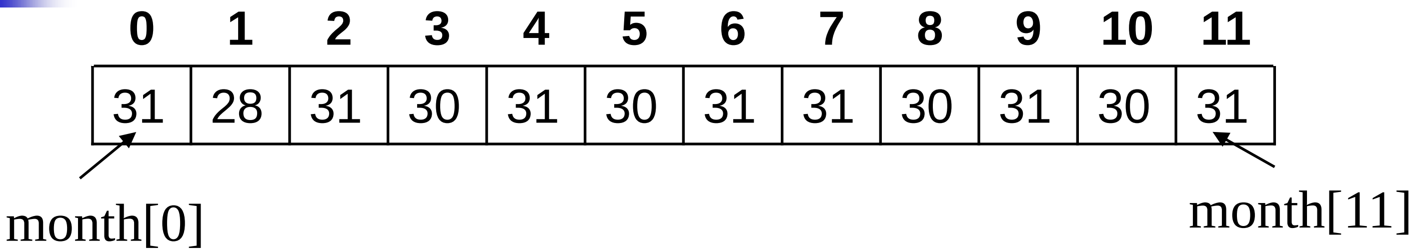
[34, 26, ~~15~~, 10]

12

List Methods

`l.append(item)`	Add item at end of list
`l.insert(pos, item)`	Add item at specified position of list
`l[pos] = value`	Replace element at pos with value
`L[start:end] = sequence`	Replace elements at pos start to end -1 with elements in sequence
`l.remove(item)`	Remove item in list
`l.pop(pos)`	Remove item at pos in list
`l.clear()`	Remove all items in list
`list(sequence)`	Convert sequence to list

Printing sequence elements



0	1	2	3	4	5	6	7	8	9	10	11
31	28	31	30	31	30	31	31	30	31	30	31

month[0]

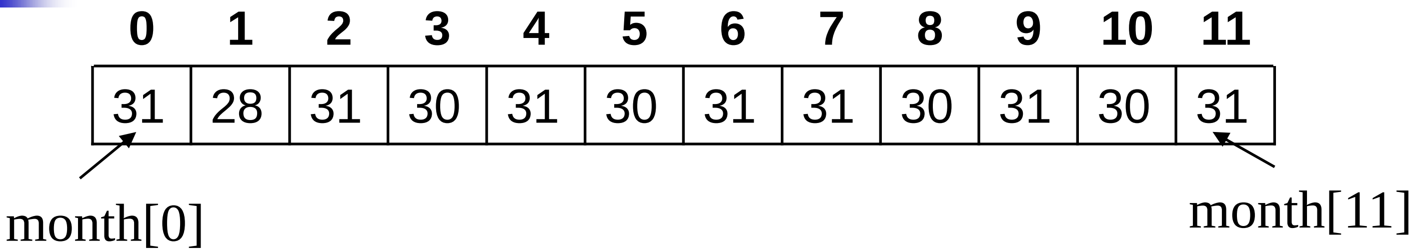
month[11]

```
month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
for index in range( 12):
    print( "Month {} has {} days.".format(index+1, month[index]))
```

Output:

```
Month 1 has 31 days.
Month 2 has 28 days.
…
```

Checking membership



0	1	2	3	4	5	6	7	8	9	10	11
31	28	31	30	31	30	31	31	30	31	30	31

month[0]

month[11]

```
month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
if 29 in month:
    print('Leap year')
else:
    print('Not a leap year')
```

Output: Not a leap year

Months with the most days

```
month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
monthName = ('January', 'February', 'March', 'April', 'May', 'June',
'July', 'August', 'September', 'October', 'November', 'December')
maxDays = max(month)
for index in range( 12):
    if month[index] == maxDays:
        print(monthName[index])
```

Output:

```
January
March
...
```

List Comprehension

Syntax:

```
    [expression for item in sequence if condition]
```

Interpreted as

```
    for item in sequence:
        if condition:
            expression
```

Example:

```
maxMonths = [monthName[index] for index in range( 12)
if month[index] == max(month)]
```

List Comprehension

```
month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
monthName = ('January', 'February', 'March', 'April', 'May', 'June',
'July', August', 'September', 'October', 'November', 'December')
maxMonths = [monthName[index] for index in range( 12)
if month[index] == max(month)]
for m in maxMonths :
    print(m)
```

Output:

```
January
March
...
```

Statistics with Lists

```
def main():
    data = getNumbers()
    xbar = mean(data)
    std = stdDev(data, xbar)
    print("\nThe mean is", xbar)
    print("The standard deviation is", std)

def getNumbers():
    nums = []
    xStr = input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        nums.append(float(xStr))
        xStr = input("Enter a number (<Enter> to quit)
  >> ")
    return nums
```

Statistics with Lists

```
def mean(nums):
    sum = 0.0
    for num in nums:
        sum = sum + num
    return sum / len(nums)
```

$$\overline{X} = \frac{\sum_{i=1}^{n} x_i}{n}$$

```
def stdDev(nums, xbar):
    sumDevSq = 0.0
    for num in nums:
        dev = xbar - num
        sumDevSq = sumDevSq + dev * dev
    return sqrt(sumDevSq/(len(nums)-1))
```

$$S = \sqrt{\frac{\Sigma\,(x - \bar{x})^2}{n - 1}}$$

Scope of Variables

- Each function is a little subprogram.
  - Variables in a function are *local*
  - *scope* - places a variable can be referenced

```
def getMonthName(i):
    monthName = ('January', 'February', 'March', 'April', 'May',
'June', 'July', August', 'September', 'October', 'November',
'December')
    return monthName[i]
```

```
print(monthName[2])
```

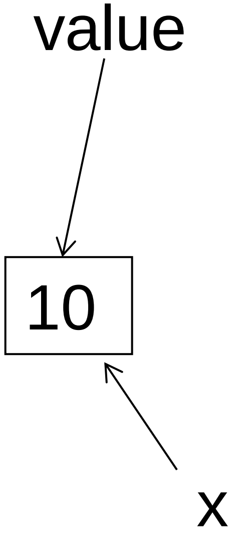
NOT OK

Passing Immutable Values

```
def double(value):
    value = 2 * value
    z = value
    return z

def main():
    x = 10
    y = double(x)
    print(x, y)

main()
```



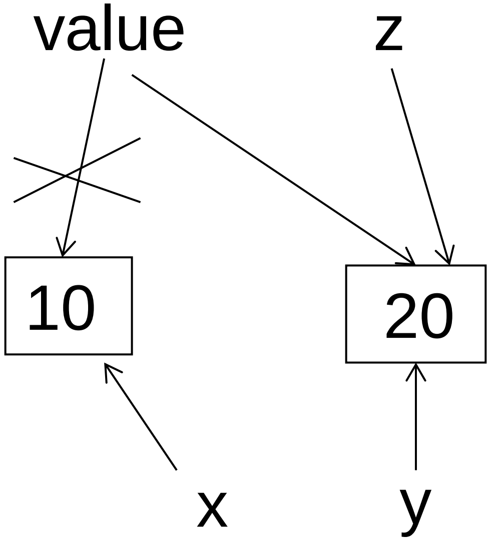
Output: ?

Passing Immutable Values

```
def double(value):
    value = 2 * value
    z = value
    return z

def main():
    x = 10
    y = double(x)
    print(x, y)

main()
```



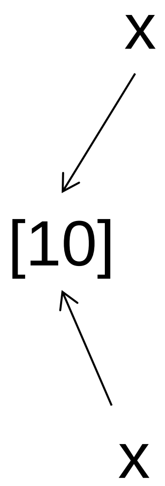
Output: ?

Passing Mutable Values

```
def double(x):
    x =  x.append(2)
    z = x
    return z

def main():
    x = [10]
    y = double(x)
    print(x, y)

main()
```



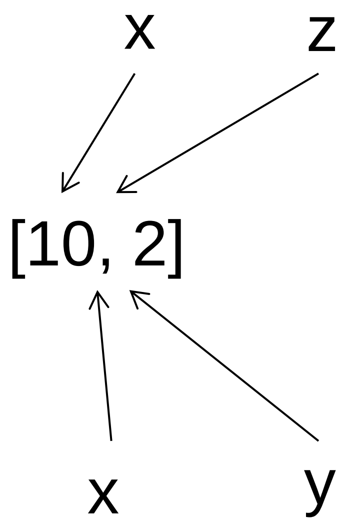
Output: ?

Passing Mutable Values

```
def double(x):
    x =  x.append(2)
    z = x
    return z

def main():
    x = [10]
    y = double(x)
    print(x, y)

main()
```



x z

[10, 2]

x y

Output: ?

Top-Down Design

- Express complex problem in terms of smaller, simpler problems repeatedly
- Continues until the problems are trivial to solve
- Put together the pieces as a solution to the original problem
- Each piece of solution is a function

Scenario

- Guess the value of a dice
- Only 3 tries
- Dice value revealed after 3 tries
- After each game, prompt the player whether he wishes to play another game

Top-Level Design

- The algorithm for the guess dice game:

```
Loop when player wants to play a game
    Roll a dice
    Play guessing game
    Ask whether player wishes to play another game
```

- Ignore whatever we don’t know how to do first.

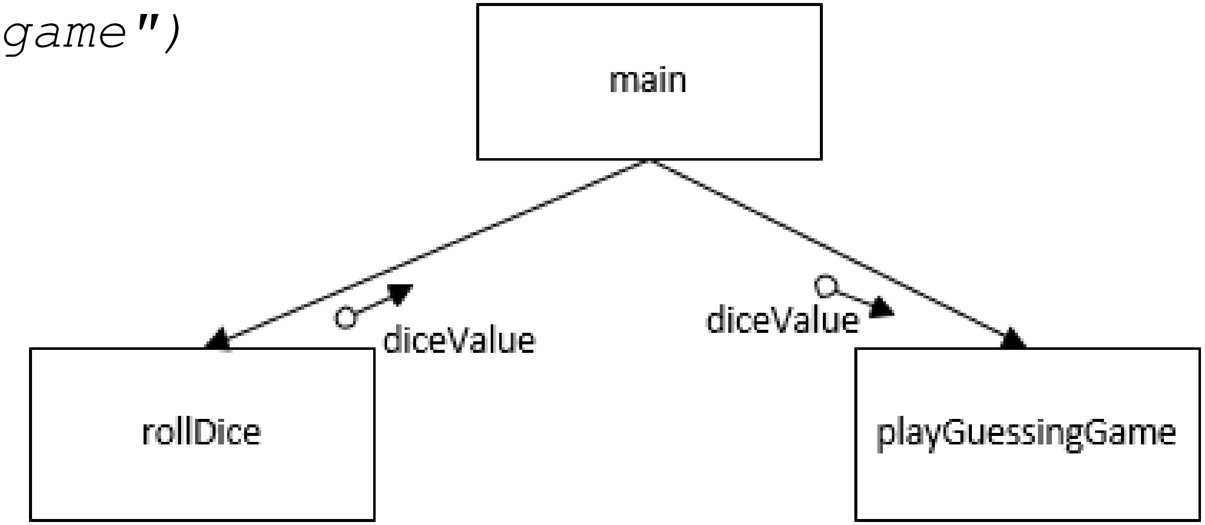
Top-Level Design

- Roll a dice
  - `rollDice` function
  - No need input from caller but give output to caller – the face value of the dice

- Play guessing game
  - `playGuessingGame` function
  - Input from caller: dice face value, but does not give output to caller

Top-Level Design

```
def main():
    playAgain = 'y'
    while playAgain[0].lower() in 'yY':
        diceValue = rollDice()
        playGuessingGame(diceValue)
        playAgain = input("Continue? y/n: ")
    print("End game")
```



Second-Level Design

- `rollDice` function - straightforward

```
from random import randint
def rollDice():
    return randint(1, 6)
```

- Implementation can change without affecting caller

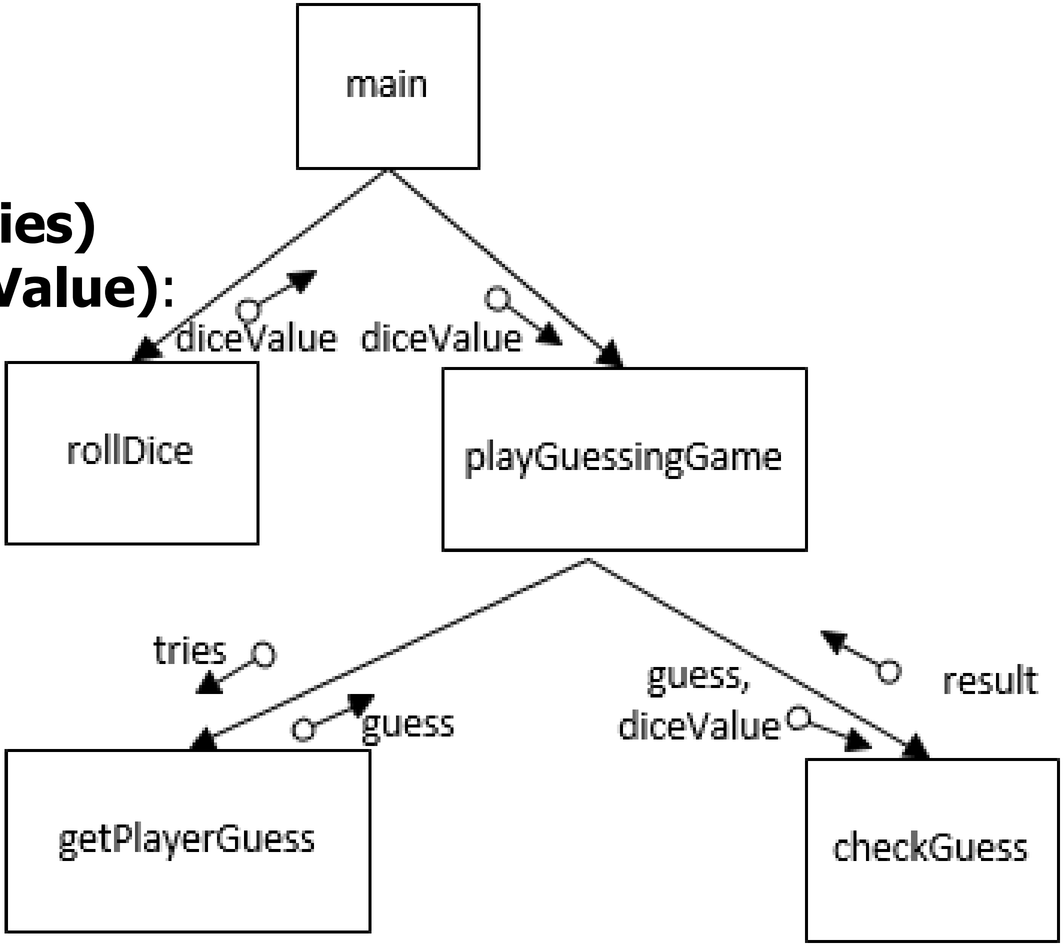
Second-Level Design

- `playGuessingGame` function – complex
- Repeat the top-down design process

```
loop 3 times
    get player’s guess
    if correct guess
       exit loop
if incorrect at end of 3 trues
    print dice value
```

Second-Level Design

```
def playGuessingGame(diceValue):
    for tries in range(1, 4):
        guess = getPlayerGuess(tries)
        if checkGuess(guess, diceValue):
            break
    else:
        print("Sorry, value is {}".format(diceValue))
```



Third-Level Design

- `getPlayerGuess` function is straightforward

```
def getPlayerGuess(tries):
    return int(input("Try {}. Enter guess: ".format(tries)))
```

- `checkGuess` function is straightforward

```
def checkGuess(guess, diceValue):
    success = diceValue == guess
    if success:
        print("You got it!")
    else:
        print("Incorrect")
    return success
```

Complete Program

```
from random import randint
def rollDice():
    return randint(1, 6)

def getPlayerGuess(tries):
    return int(input("Try {}. Enter guess: ".format(tries)))

def checkGuess(guess, diceValue):
    success = diceValue == guess
    if success:
        print("You got it!")
    else:
        print("Incorrect")
    return success
```

```
def playGuessingGame(diceValue):
    for tries in range(1,4):
        guess = getPlayerGuess(tries)
        if checkGuess(guess, diceValue):
            break
    else:
        print("Sorry, value is {}".format(diceValue))

def main():
    playAgain = 'y'
    while playAgain[0].lower() in 'yY':
        diceValue = rollDice()
        playGuessingGame(diceValue)
        playAgain = input("Continue? y/n: ")
    print("End game")
```

Bottom-Up Implementation and Unit testing

- Bottom-Up Implementation
  - Implement the functions at the lowest level of the structure chart
- Unit Testing
  - Start at the lowest levels of the structure, testing each component as it is complete
  - Systematically test the implementation