ESTRUCTURA DE DATOS

PROFESOR ISAAC RUDOMÍN

22P

PYTHON PROGRAMMING EXPRESS

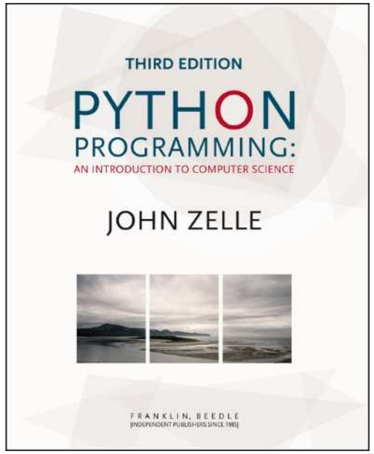
MATERIAL BASADO EN CURSO DE JOHN ZELLE

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA
Unidad Cuajimalpa

Introduction

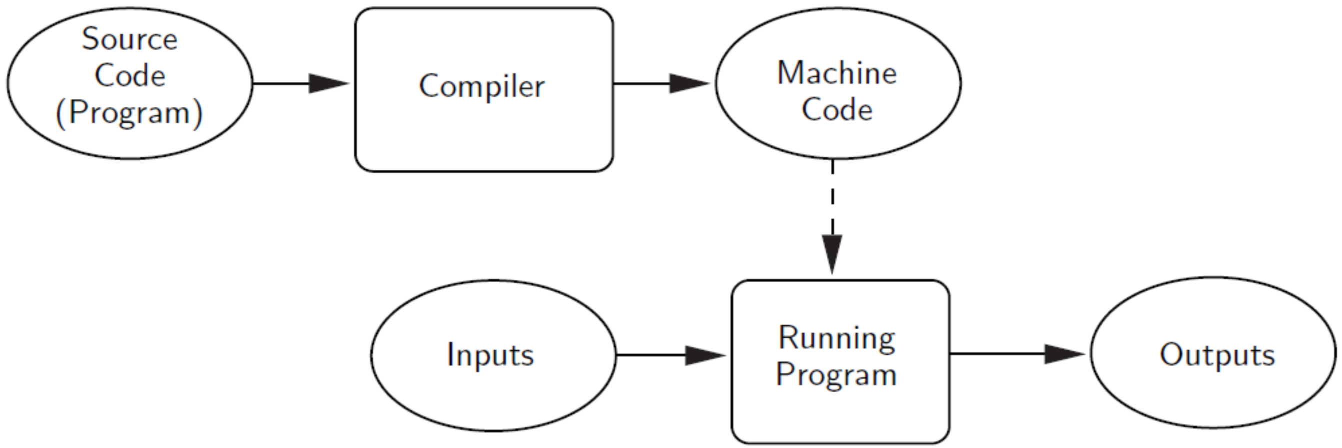
Homepage

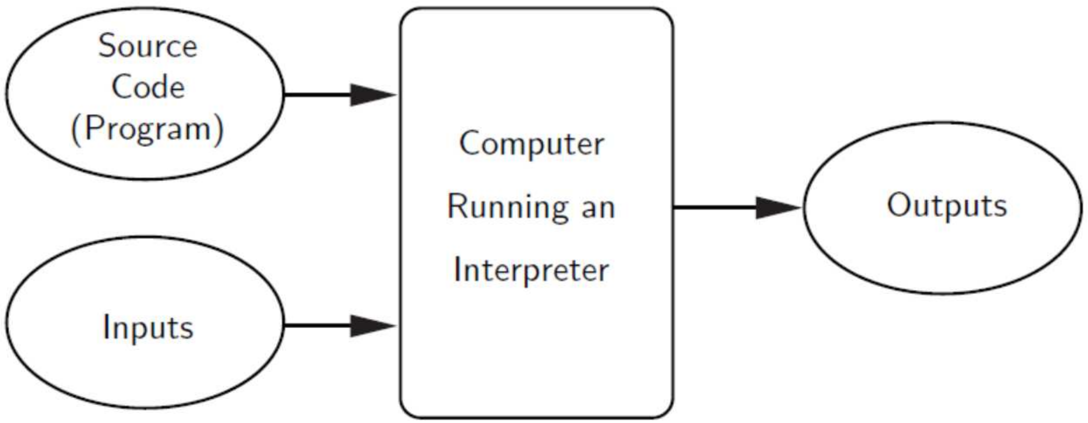
Books



The Magic of Python

Compiled vs Interpreted Programming Languages





The Magic of Python

- The “>>>” is a Python *prompt* indicating that Python is ready for us to give it a command. These commands are called *statements*.

```
>>> print("Hello, world")
Hello, world
>>> print(2+3)
5
>>> print("2+3=", 2+3)
2+3= 5
>>>
```

- Usually we want to execute several statements together that solve a common problem. One way to do this is to use a function.

```
>>> def hello():
        print("Hello")
        print("Computers are Fun")

>>>
```

The Magic of Python

```
# File: chaos.py
# A simple program illustrating chaotic behavior

def main():
    print("This program illustrates a chaotic function")
    x = eval(input("Enter a number between 0 and 1: "))
    for i in range(10):
        x = 3.9 * x * (1 - x)
        print(x)

main()
```

- We'll use *filename.py* when we save our work to indicate it's a Python program.
- In this code we're defining a new function called **main**.
- The main() at the end tells Python to run the code.

Example Program: Temperature Converter

```
#convert.py
# A program to convert Celsius temps to Fahrenheit
# by: Susan Computewell

def main():
    celsius = eval(input("What is the Celsius
  temperature? "))
    fahrenheit = (9/5) * celsius + 32
    print("The temperature is ",fahrenheit," degrees
  Fahrenheit.")

main()
```

Simultaneous Assignment

We can swap the values of two variables quite easily in Python! `x, y = y, x`

```
>>> x = 3
>>> y = 4
>>> print(x, y)
3 4
>>> x, y = y, x
>>> print(x, y)
4 3
```

Definite Loops

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

The beginning and end of the body are indicated by indentation.

```
>>> for i in [0,1,2,3]:
            print (i)
>>> for odd in [1, 3, 5, 7]:
            print(odd*odd)
>>>
```

Example Program: Future Value

```
# futval.py
#    A program to compute the value of an investment
#    carried 10 years into the future


def main():
    print("This program calculates the future value of a 10-year investment.")


    principal = eval(input("Enter the initial principal: "))
    apr = eval(input("Enter the annual interest rate: "))


    for i in range(10):
        principal = principal * (1 + apr)


    print ("The value in 10 years is:", principal)


main()
```

Numeric Data Types

- Python has a special function to tell us the data type of any value.
- Operations on ints produce ints, operations on floats produce floats (except for /).

```
>>> type(3)
<class 'int'>
>>> type(3.1)
<class 'float'>
>>> type(3.0)
<class 'float'>
>>> myInt = 32
>>> type(myInt)
<class 'int'>
>>>
```

```
>>> 3.0+4.0
7.0
>>> 3+4
7
>>> 3.0*4.0
12.0
>>> 3*4
12
>>> 10.0/3.0
3.3333333333333335
>>> 10/3
3.3333333333333335
>>> 10 // 3
3
>>> 10.0 // 3.0
3.0
```

```
>>> float(22//5)
4.0
>>> int(4.5)
4
>>> int(3.9)
3
>>> round(3.9)
4
>>> round(3)
3
>>> round(3.1415926,
   2)
3.14
```

Type Conversions & Rounding

```
# change.py
#   A program to calculate the value of some change in dollars

def main():
    print("Change Counter")
    print()
    print("Please enter the count of each coin type.")
    quarters = int(input("Quarters: "))
    dimes = int(input("Dimes: "))
    nickels = int(input("Nickels: "))
    pennies = int(input("Pennies: "))
    total = quarters * .25 + dimes * .10 + nickels * .05 + pennies * .01
    print()
    print("The total value of your change is", total)
```

Using the Math Library

```
# quadratic.py
#    A program that computes the real roots of a quadratic equation.
#    Illustrates use of the math library.
#    Note: This program crashes if the equation has no real roots.

import math  # Makes the math library available.

def main():
    print("This program finds the real solutions to a quadratic")
    print()

    a, b, c = eval(input("Please enter the coefficients (a, b, c): "))

    discRoot = math.sqrt(b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)

    print()
    print("The solutions are:", root1, root2 )
```

Using the Math Library

Python	Mathematics	English
pi	π	An approximation of pi
e	e	An approximation of e
sqrt(x)	$\sqrt{x}$	The square root of x
sin(x)	$\sin x$	The sine of x
cos(x)	$\cos x$	The cosine of x
tan(x)	$\tan x$	The tangent of x
asin(x)	$\arcsin x$	The inverse of sine x
acos(x)	$\arccos x$	The inverse of cosine x
atan(x)	$\arctan x$	The inverse of tangent x

Python	Mathematics	English
log(x)	$\ln x$	The natural (base *e*) logarithm of *x*
log10(x)	$\log_{10} x$	The common (base 10) logarithm of *x*
exp(x)	e^x	The exponential of x
ceil(x)	$\lceil x \rceil$	The smallest whole number >= *x*
floor(x)	$\lfloor x \rfloor$	The largest whole number <= *x*

Accumulating Results: Factorial

```
# factorial.py
#    Program to compute the factorial of a number
#    Illustrates for loop with an accumulator

def main():
    n = eval(input("Please enter a whole number: "))
    fact = 1
    for factor in range(n,1,-1):
       fact = fact * factor
    print("The factorial of", n, "is", fact)

main()
```

The Limits of Int

- What is 100!?

```
>>> main()
Please enter a whole
   number: 100
The factorial of 100 is
   9332621544394415268169923
   8856266700490715968264381
   6214685929638952175999932
   2991560894146397615651828
   6253697920827223758251185
   2109168640000000000000000
   00000000
```

- Wow! That’s a pretty big number!

- Floats are approximations
- Floats allow us to represent a larger range of values, but with fixed precision.
- Python has a solution, expanding ints!
- Python ints are not a fixed size and expand to handle whatever value it holds.

The String Data Type

```
>>> str1="Hello"
>>> str2='spam'
>>> print(str1, str2)
Hello spam
>>> type(str1)
<class 'str'>
>>> type(str2)
<class 'str'>
```

The String Data Type

Getting a string as input

```
>>> firstName = input("Please enter your name: ")
Please enter your name: John
>>> print("Hello", firstName)
Hello John
```

Notice that the input is not `eval`uated. We want to store the typed characters, not to evaluate them as a Python expression.

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet = "Hello Bob"
>>> greet[0]
'H'
>>> print(greet[0], greet[2], greet[4])
H l o
>>> x = 8
>>> print(greet[x - 2])
B
```

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

In a string of *n* characters, the last character is at position *n-1* since we start counting with 0.

We can index from the right side using negative indexes.

```
>>> greet[-1]
'b'
>>> greet[-3]
'B'
```

The String Data Type

Slicing:

`<string>[<start>:<end>]`

- `start` and `end` should both be ints
- The slice contains the substring beginning at position `start` and runs up to **but doesn't include** the position `end`.

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet[0:3]
'Hel'
>>> greet[5:9]
' Bob'
>>> greet[:5]
'Hello'
>>> greet[5:]
' Bob'
>>> greet[:]
'Hello Bob'
```

The String Data Type

- The function *len* will return the length of a string.

```
>>> "spam" + "eggs"
'spameggs'
>>> "Spam" + "And" + "Eggs"
'SpamAndEggs'
>>> 3 * "spam"
'spamspamspam'
>>> "spam" * 5
'spamspamspamspamspam'
>>> (3 * "spam") + ("eggs" * 5)
'spamspamspameggseggseggseggseggs'
```

```
>>> len("spam")
4
>>> for ch in "Spam!":
            print (ch, end=" ")

S p a m !
```

The String Data Type

Operator	Meaning
+	Concatenation
*	Repetition
<string>[]	Indexing
<string>[:]	Slicing
len(<string>)	Length
for <var> in <string>	Iteration through characters

Simple String Processing

Usernames on a computer system

- First initial, first seven characters of last name

```
# get user’s first and last names
first = input("Please enter your first name (all lowercase): ")
last = input("Please enter your last name (all lowercase): ")

# concatenate first initial with 7 chars of last name
uname = first[0] + last[:7]
```

Simple String Processing

```
# month.py
#  A program to print the abbreviation of a month, given its number

def main():

    # months is used as a lookup table
    months = "JanFebMarAprMayJunJulAugSepOctNovDec"

    n = int(input("Enter a month number (1-12): "))

    # compute starting position of month n in months
    pos = (n-1) * 3

    # Grab the appropriate slice from months
    monthAbbrev = months[pos:pos+3]

    # print the result
    print ("The month abbreviation is", monthAbbrev + ".")
```

Lists as Sequences

- Strings are always sequences of characters, but *lists* can be sequences of arbitrary values.
- Lists can have numbers, strings, or both!

```
myList = [1, "Spam ", 4, "U"]
```

Lists as Sequences

```
# month2.py
#  A program to print the month name, given it's number.
#  This version uses a list as a lookup table.

def main():

    # months is a list used as a lookup table
    months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

    n = int(input("Enter a month number (1-12): "))

    print ("The month abbreviation is", months[n-1] + ".")
```

- Note that the months line overlaps a line. Python knows that the expression isn't complete until the closing ']' is encountered.

Lists as Sequences

```
# month2.py
#  A program to print the month name, given it's number.
#  This version uses a list as a lookup table.

def main():
    # months is a list used as a lookup table
    months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
    n = int(input("Enter a month number (1-12): "))

    print ("The month abbreviation is", months[n-1] + ".")
```

- Since the list is indexed starting from 0, the *n-1* calculation is straight-forward enough to put in the print statement without needing a separate step.

Lists as Sequences

This version of the program is easy to extend to print out the whole month name rather than an abbreviation!

```
months = ["January", "February", "March",
          "April", "May", "June", "July",
          "August", "September", "October",
          "November", "December"]
```

Lists as Sequences

- Lists are *mutable*, meaning they can be changed. Strings can **not** be changed.

```
>>> myList = [34, 26, 15, 10]
>>> myList[2]
15
>>> myList[2] = 0
>>> myList
[34, 26, 0, 10]
>>> myString = "Hello World"
>>> myString[2]
'l'
>>> myString[2] = "p"

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    myString[2] = "p"
TypeError: object doesn't support item assignment
```

More String Methods

- `s.capitalize()`

– Copy of s with only the first character capitalized

- `s.title()`

– Copy of s; first character of each word capitalized

- `s.center(width)`

– Center s in a field of given width

- `s.count(sub)`

– Count the number of occurrences of sub in s

- `s.find(sub)`

– Find the first position where sub occurs in s

- `s.join(list)`

– Concatenate list of strings into one large string using s as separator.

- `s.ljust(width)`

– Like center, but s is left-justified

More String Methods

- `s.lower()`
  – Copy of s in all lowercase letters
- `s.lstrip()`
  – Copy of s with leading whitespace removed
- `s.replace(oldsub, newsub)`
  – Replace occurrences of oldsub in s with newsub
- `s.rfind(sub)`
  – Like find, but returns the right-most position
- `s.rjust(width)`
  – Like center, but s is right-justified
- `s.rstrip()`
  – Copy of s with trailing whitespace removed
- `s.split()`
  – Split s into a list of substrings
- `s.upper()`
  – Copy of s; all characters converted to uppercase

Lists Have Methods, Too

- The `append` method can be used to add an item at the end of a list.
- `squares = []`
- `for x in range(1,101):`
```
  squares.append(x*x)
```

We start with an empty list ([]) and each number from 1 to 100 is squared and appended to it (`[1, 4, 9, …, 10000]`).

Lists Have Methods, Too

```
# numbers2text2.py
#     A program to convert a sequence of Unicode numbers into
#         a string of text. Efficient version using a list accumulator.

def main():
    print("This program converts a sequence of Unicode numbers into")
    print("the string of text that it represents.\n")

    # Get the message to encode
    inString = input("Please enter the Unicode-encoded message: ")

    # Loop through each substring and build Unicode message
    chars = []
    for numStr in inString.split():
        codeNum = int(numStr)             # convert digits to a number
        chars.append(chr(codeNum))         # accumulate new character

    message = "".join(chars)
    print("\nThe decoded message is:", message)
```

Input/Output as String Manipulation

We now have a complete set of type conversion operations:

Function	Meaning
float(<expr>)	Convert expr to a floating point value
int(<expr>)	Convert expr to an integer value
str(<expr>)	Return a string representation of expr
eval(<string>)	Evaluate string as an expression

Defining Functions

Functions and Parameters: The Details

- A function definition looks like this:

```
def <name>(<formal-parameters>):
    <body>
```

- The name of the function must be an identifier
- Formal-parameters is a (possibly empty) list of variable names
- A function is called by using its name followed by a list of *actual parameters* or *arguments*.

```
<name>(<actual-parameters>)
```

Functions That Return Values

- This function returns the square of a number:

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

- When Python encounters `return`, it exits the function and returns control to the point where the function was called.
- In addition, the value(s) provided in the `return` statement are sent back to the caller as an expression result.
- We can use the square function to write a routine to calculate the distance between (x_1,y_1) and (x_2,y_2).

```
def distance(p1, p2):
    dist = math.sqrt(square(p2.getX() - p1.getX()) +
                     square(p2.getY() - p1.getY()))
    return dist
```

Functions That Return Values

- Sometimes a function needs to return more than one value.
- To do this, simply list more than one expression in the `return` statement.

```
def sumDiff(x, y):
    sum = x + y
    diff = x – y
    return sum, diff
```

- When calling this function, use simultaneous assignment.

```
num1, num2 = eval(input("Enter two numbers (num1, num2) "))
s, d = sumDiff(num1, num2)
print("The sum is", s, "and the difference is", d)
```

- As before, the values are assigned based on position, so s gets the first value returned (the sum), and d gets the second (the difference).

Functions That Return Values

- One "gotcha" – all Python functions return a value, whether they contain a `return` statement or not. Functions without a `return` hand back a special object, denoted `None`.
  - A common problem is writing a value-returning function and omitting the `return`!
- The formal parameters of a function only receive the *values* of the actual parameters. The function does not have access to the variable that holds the actual parameter.
  - Python is said to pass all parameters *by value*.
- Some programming languages (C++, Ada, and many more) do allow variables themselves to be sent as parameters to a function. This mechanism is said to pass parameters *by reference*.
  - When a new value is assigned to the formal parameter, the value of the variable in the calling program actually changes.

Functions that Modify Parameters

- Since Python doesn't have this capability, we can program the `addInterest` function so that it returns the `newBalance`.
- When `addInterest` terminates, the list stored in `amounts` now contains the new values.
- The variable `amounts` wasn't changed (it's still a list), but the state of that list has changed, and this change is visible to the calling program.

```
# addinterest3.py
#    Illustrates modification of a mutable parameter (a list).

def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1+rate)

def test():
    amounts = [1000, 2200, 800, 360]
    rate = 0.05
    addInterest(amounts, 0.05)
    print(amounts)

test()
```

Decision Structures

Decisions & Loops

- The `if` statement is used to implement the decision.

```
if <condition>:
      <body>
```

- The body is a sequence of one or more statements indented under the `if` heading.

- The `if-else` statement implements two-way decision:

```
if <condition>:
    <statements>
else:
    <statements>
```

- The `if-elif-else` statement implements multiway decision :

```
if <condition1>:
   <case1 statements>
elif <condition2>:
   <case2 statements>
elif <condition3>:
   <case3 statements>
…
else:
   <default statements>
```

Forming Simple Conditions

Python	Mathematics	Meaning
<	$<$	Less than
<=	$\leq$	Less than or equal to
==	$=$	Equal to
>=	$\geq$	Greater than or equal to
>	$>$	Greater than
!=	$\neq$	Not equal to

Multi-Way Decisions

```
# quadratic4.py
import math

def main():
    print("This program finds the real solutions to a quadratic\n")

    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))

    discrim = b * b - 4 * a * c
    if discrim < 0:
        print("\nThe equation has no real roots!")
    elif discrim == 0:
        root = -b / (2 * a)
        print("\nThere is a double root at", root)
    else:
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2 )
```

Exception Handling

- The `try` statement has the following form:

```
try:
   <body>
except <ErrorType>:
   <handler>
```

- When Python encounters a `try` statement, it attempts to execute the statements inside the body.
- If there is no error, control passes to the next statement after the `try`…`except`.

Exception Handling

```
# quadratic5.py
import math

def main():
    print ("This program finds the real solutions to a quadratic\n")

    try:
        a = float(input("Enter coefficient a: "))
        b = float(input("Enter coefficient b: "))
        c = float(input("Enter coefficient c: "))
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2)
    except ValueError:
        print("\nNo real roots")
```

Exception Handling

```
# quadratic6.py
import math

def main():
    print("This program finds the real solutions to a quadratic\n")

    try:
        a = float(input("Enter coefficient a: "))
        b = float(input("Enter coefficient b: "))
        c = float(input("Enter coefficient c: "))
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2 )
    except ValueError as excObj:
        if str(excObj) == "math domain error":
            print("No Real Roots")
        else:
            print("Invalid coefficient given.")
    except:
        print("\nSomething went wrong, sorry!")
```

MAXN

```
# program: maxn.py
#   Finds the maximum of a series of numbers

def main():
    n = int(input("How many numbers are there? "))

    # Set max to be the first value
    max = float(input("Enter a number >> "))

    # Now compare the n-1 successive values
    for i in range(n-1):
        x = float(input("Enter a number >> "))
        if x > max:
            max = x

    print("The largest value is", max)
```

- Or use Python's built-in function called `max` that returns the largest of its parameters.

- ```
  def main():
      x1, x2, x3 = eval(input("Please enter three values: "))
      print("The largest value is", max(x1, x2, x3))
  ```
  ```

For Loops

- 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.

```
# average1.py
#    A program to average a set of numbers
#    Illustrates counted loop with accumulator

def main():
    n = int(input("How many numbers do you have? "))
    sum = 0.0
    for i in range(n):
        x = float(input("Enter a number >> "))
        sum = sum + x
    print("\nThe average of the numbers is", sum / n)
```

Indefinite Loops

- ```
  while <condition>:
      <body>
  ```
- `condition` is a Boolean expression, just like in `if` statements. The body is a sequence of one or more statements.
- Semantically, the body of the loop executes repeatedly as long as the condition remains true. When the condition is false, the loop terminates.
  ```

Sentinel Loops

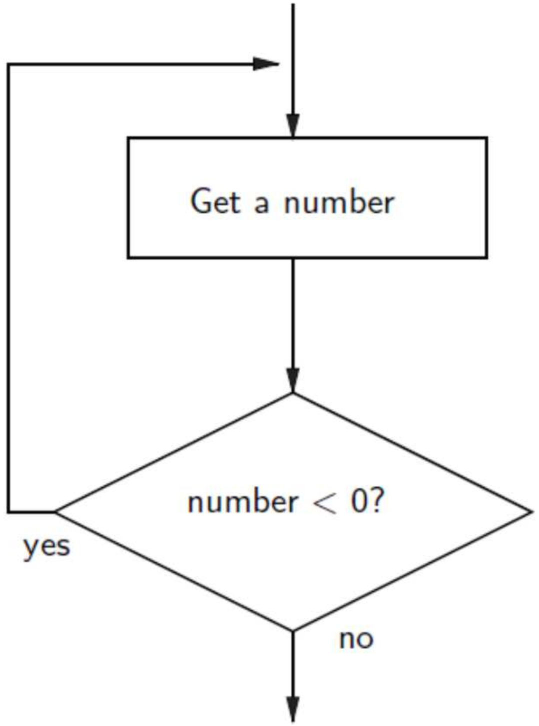
```
# average4.py
#    A program to average a set of numbers
#    Illustrates sentinel loop using empty string as sentinel

def main():
    sum = 0.0
    count = 0
    xStr = input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = float(xStr)
        sum = sum + x
        count = count + 1
        xStr = input("Enter a number (<Enter> to quit) >> ")
    print("\nThe average of the numbers is", sum / count)
```

Boolean Operators

- The Boolean operators `and` and `or` are used to combine two Boolean expressions and produce a Boolean result.
- `<expr> and <expr>`
- `<expr> or <expr>`

Post-Test Loop

```
repeat
    get a number from the user
until number is >= 0
```



- When the condition test comes after the body of the loop it's called a *post-test loop*.
- A post-test loop always executes the body of the code at least once.
- Python doesn't have a built-in statement to do this, but we can do it with a slightly modified `while` loop.

Post-Test Loop

- We seed the loop condition so we're guaranteed to execute the loop once.

```
number = -1        # start with an illegal value
while number < 0: # to get into the loop
    number = float(input("Enter a positive number: "))
```

- By setting `number` to –1, we force the loop body to execute at least once.

- The same algorithm implemented with a `break`:

```
while True:
    number = float(input("Enter a positive number: "))
    if x >= 0: break # Exit loop if number is valid
```

- A while loop continues as long as the expression evaluates to true. Since `True` *always* evaluates to true, it looks like an infinite loop!

Post-Test Loop

- In the `while` loop version, this is awkward:

```
number = -1
while number < 0:
    number = float(input("Enter a positive number: "))
    if number < 0:
        print("The number you entered was not positive")
```

- We're doing the validity check in two places!

- Adding the warning to the `break` version only adds an `else` statement:

```
while True:
    number = float(input("Enter a positive number: "))
    if x >= 0:
        break # Exit loop if number is valid
    else:
        print("The number you entered was not positive.")
```

Loop and a Half

- Stylistically, some programmers prefer the following approach:

```
while True:
    number = float(input("Enter a positive number: "))
    if x >= 0: break # Loop exit
    print("The number you entered was not positive")
```

- Here the loop exit is in the middle of the loop body. This is what we mean by a *loop and a half*.

- The loop and a half is an elegant way to avoid the priming read in a sentinel loop.

```
while True:
     get next data item
     if the item is the sentinel: break
     process the item
```

- This method is faithful to the idea of the sentinel loop, the sentinel value is not processed!

Data Collections

Lists and Arrays

- A list or array is a sequence of items where the entire sequence is referred to by a single name (i.e. `s`) and individual items can be selected by indexing (i.e. `s[i]`).
  - In other programming languages, arrays are generally a fixed size, meaning that when you create the array, you have to specify how many items it can hold.
  - Arrays are generally also *homogeneous*, meaning they can hold only one data type.
- Python lists are dynamic. They can grow and shrink on demand.
  - Python lists are also *heterogeneous*, a single list can hold arbitrary data types.
  - Python lists are mutable sequences of arbitrary objects.

List Operations

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration
<expr> in <seq>	Membership (Boolean)

List Operations

- Except for the membership check, we've used these operations before on strings.
- The membership operation can be used to see if a certain value appears anywhere in a sequence.

```
>>> lst = [1,2,3,4]
>>> 3 in lst
True
```

- The summing example from earlier can be written like this:

```
sum = 0
for x in s:
    sum = sum + x
```

- Unlike strings, lists are mutable:

```
>>> lst = [1,2,3,4]
>>> lst[3]
4
>>> lst[3] = "Hello“
>>> lst
[1, 2, 3, 'Hello']
>>> lst[2] = 7
>>> lst
[1, 2, 7, 'Hello']
```

List Operations

- A list of identical items can be created using the repetition operator. This command produces a list containing 50 zeroes:

```
zeroes = [0] * 50
```

- Lists are often built up one piece at a time using append.

```
nums = []
x = float(input('Enter a number: '))
while x >= 0:
    nums.append(x)
    x = float(input('Enter a number: '))
```

- Here, `nums` is being used as an accumulator, starting out empty, and each time through the loop a new value is tacked on.

List Operations

Method	Meaning
<list>.append(x)	Add element x to end of list.
<list>.sort()	Sort (order) the list. A comparison function may be passed as a parameter.
<list>.reverse()	Reverse the list.
<list>.index(x)	Returns index of first occurrence of x.
<list>.insert(i, x)	Insert x into list at index i.
<list>.count(x)	Returns the number of occurrences of x in list.
<list>.remove(x)	Deletes the first occurrence of x in list.
<list>.pop(i)	Deletes the i^{th} element of the list and returns its value.

List Operations

- Most of these methods don't return a value – they change the contents of the list in some way.
- Lists can grow by appending new items, and shrink when items are deleted. Individual items or entire slices can be removed from a list using the `del` operator.
- ```
  >>> myList=[34, 26, 0, 10]
  >>> del myList[1]
  >>> myList
  [34, 0, 10]
  >>> del myList[1:3]
  >>> myList
  [34]
  ```
- `del` isn't a list method, but a built-in operation that can be used on list items.
  ```

List Operations

- Basic list principles
  - A list is a sequence of items stored as a single object.
  - Items in a list can be accessed by indexing, and sublists can be accessed by slicing.
  - Lists are mutable; individual items or entire slices can be replaced through assignment statements.
  - Lists support a number of convenient and frequently used methods.
  - Lists will grow and shrink as needed.

Non-sequential Collections

- After lists, a *dictionary* is probably the most widely used collection data type.
  - Dictionaries are not as common in other languages as lists (arrays).
  - Lists allow us to store and retrieve items from sequential collections.
  - When we want to access an item, we look it up by index – its position in the collection.
- What if we wanted to look students up by student id number? In programming, this is called a *key-value pair*
  - We access the value (the student information) associated with a particular key (student id)

Dictionary Basics

- Three are lots of examples!: Names and phone numbers, Usernames and passwords, State names and capitals
  - A collection that allows us to look up information associated with arbitrary keys is called a *mapping*.
  - Python dictionaries are *mapping*s. Other languages call them *hashes* or *associative arrays*.
- Dictionaries can be created in Python by listing key-value pairs inside of curly braces.
- Keys and values are joined by ":" and are separated with commas.

```
>>>passwd ={"guido":"superprogrammer","turing":"genius","bill":"monopoly"}
```

- We use an indexing notation to do lookups

```
>>> passwd["guido"]
'superprogrammer'
```

Dictionary Basics

- `<dictionary>[<key>]` returns the object with the associated key.
- Dictionaries are mutable.

```
>>> passwd["bill"] = "bluescreen"
>>> passwd
{'guido': 'superprogrammer', 'bill':
'bluescreen', 'turing': 'genius'}
```

- Did you notice the dictionary printed out in a different order than it was created?

Dictionary Basics

- Mappings are inherently unordered.
  - Internally, Python stores dictionaries in a way that makes key lookup very efficient.
- When a dictionary is printed out, the order of keys will look essentially random.
  - If you want to keep a collection in a certain order, you need a sequence, not a mapping!
- Keys can be any immutable type, values can be any type, including programmer-defined.

Dictionary Operations

- Like lists, Python dictionaries support a number of handy built-in operations.
- A common method for building dictionaries is to start with an empty collection and add the key-value pairs one at a time.

```
passwd = {}
for line in open('passwords', 'r'):
    user, pass = line.split()
    passwd[user] = pass
```

Dictionary Operations

Method	Meaning
<key> in <dict>	Returns true if dictionary contains the specified key, false if it doesn't.
<dict>.keys()	Returns a sequence of keys.
<dict>.values()	Returns a sequence of values.
<dict>.items()	Returns a sequence of tuples (key, value) representing the key-value pairs.
del <dict>[<key>]	Deletes the specified entry.
<dict>.clear()	Deletes all entries.
for <var> in <dict>:	Loop over the keys.
<dict>.get(<key>, <default>)	If dictionary has key returns its value; otherwise returns default.

Dictionary Operations

```
>>> list(passwd.keys())
['guido', 'turing', 'bill']
>>> list(passwd.values())
['superprogrammer', 'genius', 'bluescreen']
>>> list(passwd.items())
[('guido', 'superprogrammer'), ('turing', 'genius'), ('bill', 'bluescreen')]
>>> "bill" in passwd
True
>>> "fred" in passwd
False
>>> passwd.get('bill','unknown')
'bluescreen'
>>> passwd.get('fred','unknown')
'unknown'
>>> passwd.clear()
>>> passwd
{}
```

Algorithms

Strategy 1: Linear Search

- This strategy is called a *linear search*, where you search through the list of items one by one until the target value is found.

```
def search(x, nums):
    for i in range(len(nums)):
        if nums[i] == x: # item found, return the index value
            return i
    return -1            # loop finished, item was not in list
```

- This algorithm wasn't hard to develop, and works well for modestly-sized lists.

Strategy 2: Binary Search

```
def search(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:          # There is still a range to search
        mid = (low + high)//2 # Position of middle item
        item = nums[mid]
        if x == item:           # Found it! Return the index
            return mid
        elif x < item:          # x is in lower half of range
            high = mid - 1      #  move top marker down
        else:                   # x is in upper half of range
            low = mid + 1       #  move bottom marker up
    return -1                   # No range left to search,
                                # x is not there
```

Recursive Problem-Solving

```
Algorithm: binarySearch – search for x in nums[low]…nums[high]

mid = (low + high)//2
if low > high
   x is not in nums
elsif x < nums[mid]
   perform binary search for x in nums[low]…nums[mid-1]
else
   perform binary search for x in nums[mid+1]…nums[high]
```

- This version has no loop, and seems to refer to itself! What’s going on??

Recursive Functions

- We’ve seen previously that factorial can be calculated using a loop accumulator.
- If factorial is written as a separate function:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

Example: String Reversal

- ```
  def reverse(s):
      return reverse(s[1:]) + s[0]
  ```
- The slice `s[1:]` returns all but the first character of the string.
- We reverse this slice and then concatenate the first character (`s[0]`) onto the end.
  ```

Example: String Reversal

- ```
  def reverse(s):
      if s == "":
          return s
      else:
          return reverse(s[1:]) + s[0]
  ```
- ```
  >>> reverse("Hello")
  'olleH'
  ```

Example: Fast Exponentiation

- ```
  def recPower(a, n):
      # raises a to the int power n
      if n ==  0:
          return 1
      else:
          factor = recPower(a, n//2)
          if n%2 == 0:    # n is even
              return factor * factor
          else:           # n is odd
              return factor * factor * a
  ```
- Here, a temporary variable called *factor* is introduced so that we don't need to calculate $a^{n//2}$ more than once, simply for efficiency.
  ```

Example: Binary Search

```
def recBinSearch(x, nums, low, high):
    if low > high:              # No place left to look, return -1
        return -1
    mid = (low + high)//2
    item = nums[mid]
    if item == x:
        return mid
    elif x < item:              # Look in lower half
        return recBinSearch(x, nums, low, mid-1)
    else:                       # Look in upper half
        return recBinSearch(x, nums, mid+1, high)
```

- We can then call the binary search with a generic search wrapping function:

```
def search(x, nums):
    return recBinSearch(x, nums, 0, len(nums)-1)
```

Recursion vs. Iteration

- ```
  def loopfib(n):
      # returns the nth Fibonacci number

      curr = 1
      prev = 1
      for i in range(n-2):
          curr, prev = curr+prev, curr
      return curr
  ```
- Note the use of simultaneous assignment to calculate the new values of `curr` and `prev`.
- The loop executes only *n-2* times since the first two values have already been “determined”.
  ```

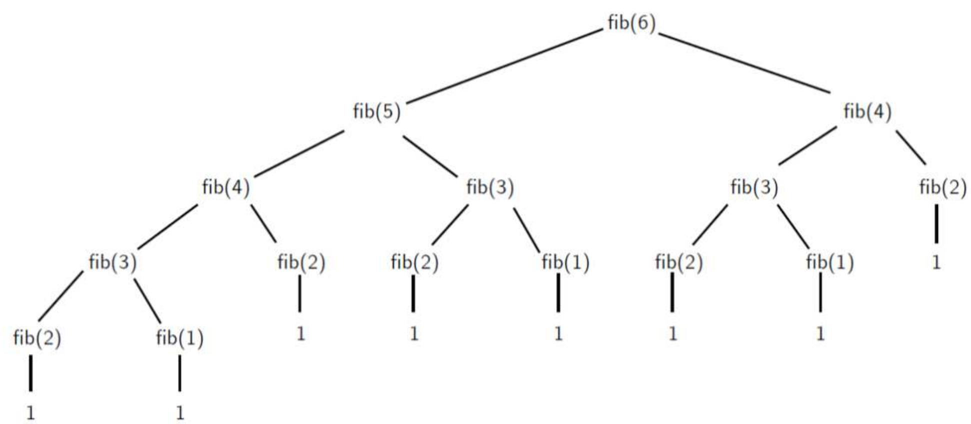
Recursion vs. Iteration

- The Fibonacci sequence also has a recursive definition:

$$fib(n) = \begin{cases} 1 & \text{if } n < 3 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

- This recursive definition can be directly turned into a recursive function!

```
def fib(n):
    if n < 3:
        return 1
    else:
        return fib(n-1)+fib(n-2))
```



- The recursive solution is extremely inefficient, since it performs many duplicate calculations!

Naive Sorting: Selection Sort

```
def selSort(nums):
    # sort nums into ascending order

    n = len(nums)

    # For each position in the list (except the very last)

    for bottom in range(n-1):
        # find the smallest item in nums[bottom]..nums[n-1]

        mp = bottom                         # bottom is smallest initially
        for i in range(bottom+1, n):        # look at each position
            if nums[i] < nums[mp]:          # this one is smaller
                mp = i                      # remember its index

        # swap smallest item to the bottom
        nums[bottom], nums[mp] = nums[mp], nums[bottom]
```

Divide and Conquer: Merge Sort

```
def merge(lst1, lst2, lst3):
    # merge sorted lists lst1 and lst2 into lst3

    # these indexes keep track of current position in each list
    i1, i2, i3 = 0, 0, 0  # all start at the front
    n1, n2 = len(lst1), len(lst2)

    # Loop while both lst1 and lst2 have more items

    while i1 < n1 and i2 < n2:
        if lst1[i1] < lst2[i2]: # top of lst1 is smaller
            lst3[i3] = lst1[i1] #  copy it into current spot in lst3
            i1 = i1 + 1
        else:                   # top of lst2 is smaller
            lst3[i3] = lst2[i2] #  copy itinto current spot in lst3
            i2 = i2 + 1
        i3 = i3 + 1             # item added to lst3, update position
```

Divide and Conquer: Merge Sort

```
# Here either lst1 or lst2 is done. One of the following loops
# will execute to finish up the merge.

   # Copy remaining items (if any) from lst1
   while i1 < n1:
       lst3[i3] = lst1[i1]
       i1 = i1 + 1
       i3 = i3 + 1

   # Copy remaining items (if any) from lst2
   while i2 < n2:
       lst3[i3] = lst2[i2]
       i2 = i2 + 1
       i3 = i3 + 1
```

Divide and Conquer: Merge Sort

```
def mergeSort(nums):
    # Put items of nums into ascending order
    n = len(nums)
    # Do nothing if nums contains 0 or 1 items
    if n > 1:
        # split the two sublists
        m = n//2
        nums1, nums2 = nums[:m], nums[m:]
        # recursively sort each piece
        mergeSort(nums1)
        mergeSort(nums2)
        # merge the sorted pieces back into original list
        merge(nums1, nums2, nums)
```

Heap Operations

- heapq implements the priority queue algorithm. Heaps are binary trees for The interesting property of a heap is that its smallest element is always the root, heap[0].

```
from heapq import heappush, heappop
heap = []
data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
for item in data:
        heappush(heap, item)
sorted = []
while heap:
     sorted.append(heappop(heap))
```

Heap Operations

- We can implement short test path using a `heapq` priority queue: See irg-astar for a similar implementation of A*:

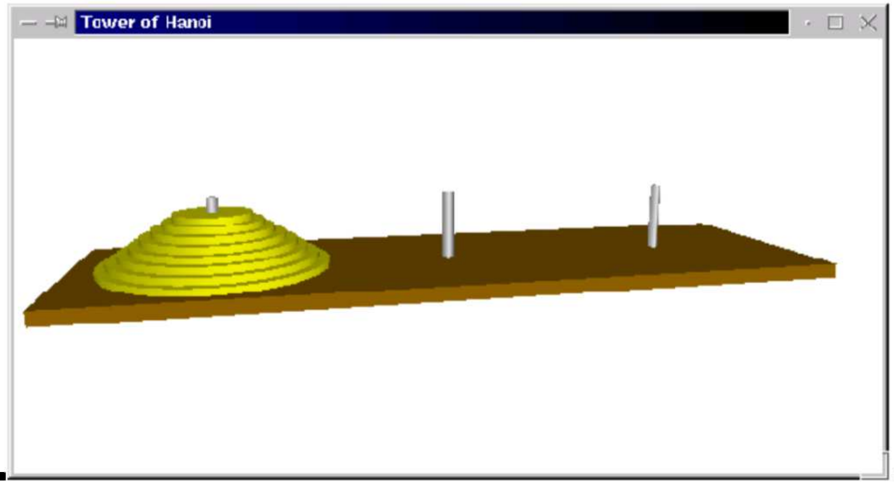
```
import heapq
import math

G = {'s':{'u':10, 'x':5},
     'u':{'v':1, 'x':2},
     'v':{'y':4},
     'x':{'u':3, 'v':9, 'y':2},
     'y':{'s':7, 'v':6}}

print ('G=', G)
def shortest_path2(G, start, end):
   def flatten(L):       # Flatten linked list of form [0,[1,[2,[]]]]
      while len(L) > 0:
         yield L[0]
         L = L[1]
```

Heap Operations

```
q = [(0, start, ())]  # Heap of (cost, path_head, path_rest).
visited = set()       # Visited vertices.
while True:
   (cost, v1, path) = heapq.heappop(q)
   if v1 not in visited:
      visited.add(v1)
      if v1 == end:
         return list(flatten(path))[::-1] + [v1]
      path = (v1, path)
      for (v2, cost2) in G[v1].items():
         if v2 not in visited:
            heapq.heappush(q, (cost + cost2, v2, path))


print ('shortest path from s to v =', shortest_path2(G,'s','v'))
```

Towers of Hanoi

– Only one disk may be moved at a time.
– A disk may not be "set aside". It may only be stacked on one of the three posts.
– A larger disk may never be placed on top of a smaller one.



Towers of Hanoi

- In `moveTower`, `n` is the size of the tower (integer), and `source`, `dest`, and `temp` are the three posts, represented by “A”, “B”, and “C”.

```
def moveTower(n, source, dest, temp):
    if n == 1:
        print("Move disk from", source, "to", dest+".")
    else:
        moveTower(n-1, source, temp, dest)
        moveTower(1, source, dest, temp)
        moveTower(n-1, temp, dest, source)
```

Towers of Hanoi

- To get things started, we need to supply parameters for the four parameters:

```
def hanoi(n):
    moveTower(n, "A", "C", "B")
```

```
>>> hanoi(3)
Move disk from A to C.
Move disk from A to B.
Move disk from C to B.
Move disk from A to C.
Move disk from B to A.
Move disk from B to C.
Move disk from A to C.
```

Towers of Hanoi

- Why is this a “hard problem”?
- How many steps in our program are required to move a tower of size n?

Number of Disks	Steps in Solution
1	1
2	3
3	7
4	15
5	31

Towers of Hanoi

- To solve a puzzle of size *n* will require $2^n - 1$ steps.
- Computer scientists refer to this as an *exponential time* algorithm.
- Exponential algorithms grow very fast.
- For 64 disks, moving one a second, round the clock, would require 580 *billion years* to complete. The current age of the universe is estimated to be about 15 billion years.