ITT 205 Problem Solving using Python

Contents

Session 1:

Python Interpreter

- An interpreter is a kind of program that executes other programs.
- When you write a Python program, the Python interpreter reads your program and carries out the instructions it contains.
- In effect, the interpreter is a layer of software logic between your code and the computer hardware on your machine.
- Depending on which flavor of Python you run, the interpreter itself may be implemented as a C program, a set of Java classes.
- Whatever form it takes, the Python code you write must always be run by this interpreter.

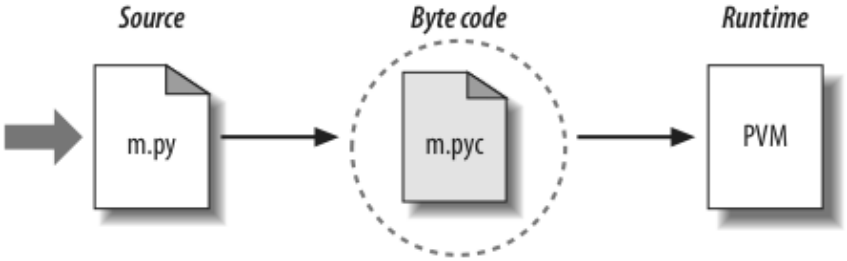
Python Byte Code

- Internally, and almost completely hidden from you, when you execute a program Python first compiles your *source code* (the statements in your file) into a format known as *byte code*.

- This byte code translation is performed to speed execution—byte code can be run much more quickly than the original source code statements in your text file.
- If the Python process has write access on your machine, it will store the byte code of your programs in files that end with a *.pyc* extension (".pyc" means compiled ".py" source).

The Python Virtual Machine (PVM)

- Once your program has been compiled to byte code (or the byte code has been loaded from existing .pyc files), it is shipped off for execution to something generally known as the Python Virtual Machine
- PVM is just a big loop that iterates through your byte code instructions, one by one, to carry out their operations
- The PVM is the runtime engine of Python; it's always present as part of the Python system, and it's the component that truly runs your scripts.



Python Implementation Alternatives

- CPython - Coded in portable ANSI C language code
- Jython - Java classes that compile Python source code to Java byte code
- IronPython - Microsoft's .NET Framework for Windows

Write a program which accepts the radius of a circle and compute the area.

```
r = float(input ("Input the radius of the circle : "))
print ("The area of the circle with radius " + str(r) + " is: " + str(3.14 * r**2))
```

Additional Reading

- Python is a programming language that lets you work quickly and integrate systems more effectively. https://www.python.org/
- What you can do with Python? Let's have a look: https://realpython.com/
- Go professional: https://www.jetbrains.com/pycharm/
- Repositories related to the Python Programming language https://github.com/python
- Where you will work? https://www.python.org/jobs/
- Coursera https://www.coursera.org/learn/python-data-analysis
- Interested in another Language? https://www.r-project.org/about.html

- Want to become a Data Scientist? http://courses.csail.mit.edu/18.337/2015/docs/50YearsDataScience.pdf
- Python do wonders. Want to See? https://numpy.org/learn/

Session 2

Python Enhancement Proposals (PEPs)

- https://www.python.org/dev/peps/

Keywords

- Keywords are the reserved words in Python.
- In Python, keywords are case sensitive.

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Identifiers

- The uppercase and lowercase letters A through Z, the underscore _ and, except for the first character, the digits 0 through 9.
- PEP 3131 -- Supporting Non-ASCII Identifiers

Values and types

A value is one of the fundamental things like a letter or a number that a program manipulates.

- >>> type("Hello, World!")
- <type 'str'>
- >>> type(17)
- <type 'int'>
- >>> type(3.2)
- <type 'float'>

Variables

- A variable is a name that refers to a value.
- >>> message = "What's up, Doc?"
- >>> n = 17
- >>> pi = 3.14159

Operators and operands

- Operators are special symbols that represent computations like addition and multiplication.

- The values the operator uses are called operands.

```
+       -       *       **      /       //      %       @
<<      >>      &       |       ^       ~       :=
<       >       <=      >=      ==      !=
```

-

Operator	Description
:=	Assignment expression
lambda	Lambda expression
if – else	Conditional expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and subtraction
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder 5
+x, -x, ~x	Positive, negative, bitwise NOT
**	Exponentiation 6
await x	Await expression
x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference
(expressions...), [expressions...], {key: value...}, {expressions...}	Binding or parenthesized expression, list display, dictionary display, set display

- In general, you cannot perform mathematical operations on strings, even if the strings look like numbers.
- Illegal expressions (assuming that message has type string):
  - message-1
  - "Hello"/123
  - message*"Hello"
  - "15"+2
  - + operator represents concatenation

    - fruit = "banana"
    - bakedGood = " nut bread"
    - print (fruit + bakedGood)
    - The * operator also works on strings; it performs repetition. For example,
    - "Fun"*3 is "FunFunFun"
-

Expressions

- An expression is a combination of values, variables, and operators

Statements

- A statement is an instruction that the Python interpreter can execute
- When you type a statement on the command line, Python executes it and displays the result, if there is one.
- A script usually contains a sequence of statements.

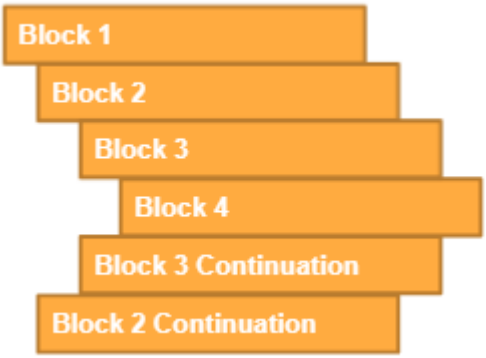
Evaluation of expressions

- Atoms are the most basic elements of expressions
- The simplest atoms are identifiers or literals
- Forms enclosed in parentheses, brackets or braces are also categorized syntactically as atoms
- The evaluation of an expression produces a value, which is why expressions can appear on the right hand side of assignment statements.
- A value all by itself is a simple expression, and so is a variable.
- Evaluating a variable gives the value that the variable refers to.
- Left-hand side of an assignment statement has to be a variable name, not an expression.
- So, the following is illegal: minute+1 = hour.
- In a script, an expression all by itself is a legal statement, but it doesn't do anything
    - 17
    - 3.2
    - "Hello, World!"
    - 1 + 1
- produces no output at all. How would you change the script to display the values of these four expressions?

Indentation

Code Blocks are defined by indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements



Indentation errors

- https://docs.python.org/2.0/ref/indentation.html

Session 3

Conditional execution

➢ if x > 0:

  print "x is positive“

- The boolean expression after the if statement is called the condition.
- If it is true, then the indented statement gets executed. If not, nothing happens.

Syntax of if statement:

- if statement is made up of a header and a block of statements
- The First unindented statement marks the end of the block.

➢ **HEADER:**

➢ **FIRST STATEMENT**

➢ **…**

➢ **LAST STATEMENT**

- There is no limit on the number of statements that can appear in the body of an if statement, but there has to be at least one.
- **You can use** pass statement if you plan to add code later

Alternative execution:

- There are two possibilities and the condition determines which one gets executed.

➢ if x%2 == 0:

  print x, "is even"

➢ else:

  print x, "is odd“

- Condition must be true or false, exactly one of the alternatives will be executed.

- The alternatives are called branches, because they are branches in the flow of execution.

Chained conditionals:

- if x < y:

  print x, "is less than", y

- elif x > y:

  print x, "is greater than", y

- else:

  print x, "and", y, "are equal“

- There is no limit of the number of elif statements

Nested conditionals:

- if x == y:

  print x, "and", y, "are equal"

- else:
  - if x < y:

    print x, "is less than", y

  - else:

    print x, "is greater than", y

  Logical operators often provide a way to simplify nested conditional statements.

- if 0 < x:
- if x < 10:
- print "x is a positive single digit."

Keyboard input:

- The input function always builds a string from the user's keystrokes and returns it to the program
- >>>first = int(input(“Enter the first number: “))
- Enter the first number: 23
- >>> second = int(input(“Enter the second number: “))
- Enter the second number: 44
- >>> print(“The sum is”, first + second)
- The sum is 67

Session 4:

For Loop:

```
for number in range(10):
    print(number)
```

```
n = 10
for i in range(4, n):
    print(i)
```

```
for number in range(0, 10, 3):  # last one is step
    print(number)
```

```
my_list = [1, 2, 3, 4, 'Python', 'is', 'neat']
for item in my_list:
    print(item)
```

break

```
my_list = [1, 2, 3, 4, 'stop', 'is', 'neat']
for item in my_list:
    if item == 'stop':
        break
    print(item)
print("After for loop")
```

continue

```
my_list = [1, 2, 3, 4, 'Python', 'is', 'neat']
for item in my_list:
    if item == 2:
        continue
    print(item)
```

Iterating over a String

```
s = "Mango"
for i in s :
    print(i)
```

Looping dictionaries

```
my_dict = {'hacker': True, 'age': 72, 'name': 'John Doe'}
for i in my_dict:
    print(i,":",my_dict[i])
```

```
for key, val in my_dict.items():
    print(key, val)
```

For Else

```
for i in range(1, 4):
        print(i)
else: # Executed because no break in for
        print("Print else statement")


for i in range(1, 4):
        print(i)
        break
else: # Not executed as there is a break
        print("No Break")
```

While loop

```
n=4
while n > 0:
  print(n)
  n = n-1
print ("Blastoff!")
```

While Else

```
i=1
while i<99:
    if i % 2 == 0:
        print ("list contains an even number")
        break
    i+=2
else:
    print ("list does not contain an even number")


while True:
  reply = input('Enter text:')
  if reply == 'stop': break
  try:
    num = int(reply)
  except:
    print('Bad Input!')
  else:
    print(int(reply) ** 2)
    print('Calculation Done')
```

Session 5

A compound data type: Strings

The bracket operator selects a single character from a string.

```
>>> fruit = "banana"
>>> letter = fruit[1]
```

The expression in brackets is called an index. An index species a member of an ordered set, in this case the set of characters in the string.

The index indicates which one you want, hence the name. It can be any integer expression

The len function returns the number of characters in a string:

```
>>> fruit = "banana"
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
length = len(fruit)
last = fruit[length] # ERROR!
```

Indices

```
length = len(fruit)
last = fruit[length-1]
```

Alternatively, we can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

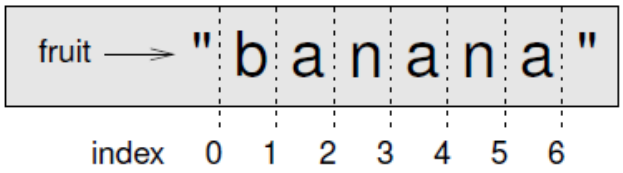
Traversal

```
index = 0
while index < len(fruit):
letter = fruit[index]
print (letter)
index = index + 1

for char in fruit:
print (char)
```

Slice

The operator `[n:m]` returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last. This behavior is counterintuitive; it makes more sense if you imagine the indices pointing *between* the characters, as in the following diagram:



If you omit the first index (before the colon), the slice starts at the beginning of the string.

If you omit the second index, the slice goes to the end of the string

String comparison

```
word = "banana"

if word == "banana":
  print ("Yes, we have bananas!")


word='banana'

if word < "banana":
  print ("Your word," + word + ", comes before banana.")
elif word > "banana":
  print ("Your word," + word + ", comes after banana.")
else:
  print ("Yes, we have bananas!")
```

All the uppercase letters come before all the lowercase letters.

```
word ='Zebra'
```

In [27]:

```
if word < "banana":
  print ("Your word," + word + ", comes before banana.")
elif word > "banana":
  print ("Your word," + word + ", comes after banana.")
else:
  print ("Yes, we have bananas!")
```

```
Your word,Zebra, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison.

Strings are immutable

```
greeting = "Hello, world!"
greeting[0]
greeting[0] = 'J'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-fbba4a7660ba> in <module>()
      1 greeting = "Hello, world!"
      2 greeting[0]
----> 3 greeting[0] = 'J'

TypeError: 'str' object does not support item assignment
```

```
greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print (newGreeting)
```

```
Jello, world!
```

```
'J' + greeting[1:]
```

```
'Jello, world!'
```

A find function: This pattern of computation is sometimes called a "eureka" traversal because as soon as we find what we are looking for, we can cry "Eureka!" and stop looking.

```
def find(str, ch):
  index = 0
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1


find('apple','l')
```

Modify the find function so that it has a third parameter,the index in the string where it should start looking

```
def find(str, ch,index):
  #
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```

Looping and counting

```
fruit = "banana"
count = 0
for char in fruit:
  if char == 'a':
    count = count + 1
print (count)
```

Session 6

Lists-List Values

```
list1= [10, 20, 30, 40]
list2= ["spam", "bungee", "swallow"]
list3= ["hello", 2.0, 5, [10, 20]]
```

```
empty = []
print(empty)
```

```
[]
```

Accessing Elements

```
numbers = [10, 20,30]
```

```
print(numbers)
```

```
[10, 20, 30]
numbers[2]
```

```
30
```

Lists are mutable

```
print (numbers)
numbers[1] = 55
print(numbers)
```

```
[10, 20, 30]
[10, 55, 30]
```

The bracket operator can appear anywhere in an expression. When it appears on the left side of an assignment, it changes one of the elements in the list, so the one-eth element of numbers, which used to be 123, is now 5.

Any integer expression can be used as an index:

```
type(3/1)
```

```
float
```

```
3/1
```

```
3.0
```

```
numbers[3/1]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-3bb47ec7ddac> in <module>()
----> 1 numbers[3/1]

TypeError: list indices must be integers or slices, not float
```

Session 7

Tuple

Syntactically, a tuple is a comma-separated list of values

```
mytuple = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is conventional to enclose tuples in parentheses:

```
tuple = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, we have to include the final comma:

```
t1 = ('a',)
type(t1)
tuple
```

Without the comma, Python treats ('a') as a string in parentheses:

```
t2 = ('a')
```

```
type(t2)
```

```
str
```

Operations on tuples are the same as the operations on lists. The index operator selects an element from a tuple.

```
tuple
```

```
('a', 'b', 'c', 'd', 'e')
```

```
tuple[-1]
```

```
'e'
```

And the slice operator selects a range of elements.

```
tuple[1:3]
```

Immutable

```
tuple
```

```
('a', 'b', 'c', 'd', 'e')
```

```
tuple[0] = 'A'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-0c3ebe4a93ea> in <module>()
----> 1 tuple[0] = 'A'

TypeError: 'tuple' object does not support item assignment
```

But if we try to modify one of the elements of the tuple, we get an error:

```
tuple = ('A',) + tuple[1:]
```

Tuple assignment

```
a=3
b=2
```

```
type(a)
int
```

```
type(b)
int
```

```
print(a,b)
3 2
(a, b) = (b, a)
```

```
type((a,b))
tuple
```

```
print(a,b)
2 3
```

Tuples as return values

```
def swap(x, y):
 return (y, x)
```

```
def fun5(x, y):
 return (y, x, p,q,r)
```

```
a,b,c,d,e = fun5(x, y)
```

Tuple Operations

```
tuple = (1,2,3,4,5,6)

tuple*3

(1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6)

tuple+tuple+tuple

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

Nested Tuple

```
NestedT =(1, 2, ("pop", "rock") ,(3,4),("disco",(1,2)))
NestedT[4][1][1]
2
print(NestedT[4][0])
disco
```

Session 8

Recursive function

A recursive function is a function that calls itself. To prevent a function from repeating itself indefinitely, it must contain at least one selection statement. This statement examines a condition called a base case to determine whether to stop or to continue with another recursive step.

Recursive functions are frequently used to design algorithms for computing values that have a recursive definition.

A recursive definition consists of equations that state what a value is for one or more base cases and one or more recursive cases. For example, the Fibonacci sequence is a series of values with a recursive definition. The first and second numbers in the Fibonacci sequence are 1. Thereafter, each number in the sequence is the sum of its two predecessors

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

If you played around with the fibonacci function , you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases very quickly. On one of our machines, fibonacci(20) finishes instantly, fibonacci(30) takes about a second, and fibonacci(40) takes roughly forever

```
import time
def fib(n):
   if n <= 1:
       return n
   else:
       return(fib(n-1) + fib(n-2))
n=int(input("Enter the value of n:"))
start_time = time.time()
print("Fibonacci(", n,")= ", fib(n))
print("Time:% ",(time.time() - start_time))
```

A call graph shows a set function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, fibonacci with n=4 calls fibonacci with n=3 and n=2. In turn, fibonacci with n=3 calls fibonacci with n=2 and n=1. And so on.

Count how many times fibonacci(0) and fibonacci(1) are called. This is an inefficient solution to the problem, and it gets far worse as the argument gets bigger.

Good solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a hint. Here is an implementation of fibonacci using hints

```
previous={0:0, 1:1}

def fibonacci(n):
    if n not in previous:
        val=fibonacci(n-1)+fibonacci(n-2)
        previous[n]=val
    return previous[n]
```

Using lambda to Create Anonymous Functions

Python includes a mechanism called lambda that allows the programmer to create functions in this manner. A lambda is an anonymous function. It has no name of its own, but contains the names of its arguments as well as a single expression. When the lambda is applied to its arguments, its expression is evaluated, and its value is returned. All of the code must appear on one line and The syntax of a lambda is very tight and restrictive: lambda<argname-1,...,argname-n>:

```
add5 = lambda x: x + 5
print(add5(7))
```

```
add5 = lambda x,y: x+5+y
print(add5(7,8))
```