Basic Python I

K.N. Toosi

M.Abdolali

Fall 2023

Hello World!
--Basic input & output--

Strings

A string is a series of characters. Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings.

- Manipulating Strings:
  - Changing Case
    - <string>.title()
    - <string>.upper()
    - <string>.lower()
- Format strings:
  - Python 3.6 & higher:
    - f"Hello, {varName}!“
  - Python 3.5 & lower
    - “Hello, {}!”.format(varName)
- Newline & tabs
  - \n \t
- Removing whitespace
  - <string>.lstrip()
  - <string>.rstrip()
  - <string>.strip()
- Concatenate strings using +
- Repeat string using *

Printing

- The print( ) function prints the specified message to the screen, or other standard output device.
- The message can be a string, or any other object, the object will be converted into a string before written to the screen.
- Seriously starting Python (or any other programming language) journey by learning about print( ):

print(“Hello World!”)

Optional but useful parameters of printing

sep='*separator*'	Optional. Specify how to separate the objects, if there is more than one. Default is ' '
end='*end*'	Optional. Specify what to print at the end. Default is '\n' (line feed)

Input

- Most programs are written to solve an end user's problem. To do so, you usually need to get some information from the user.
- In Python, we can get input from the user using a built-in function called input.
- Here's how it is used, most typically in an assignment statement:

```
<variable> = input(<prompt string>)
```

- On the right side of the equals sign is the call to the input built-in function. When you make the call, you must pass in a prompt string, which is any string that you want the user to see. The prompt is a question you want the user to answer. The input function returns all the characters that the user types, as a string.

How the input( ) Function Works

- The input( ) function pauses your program and waits for the user to enter some text. Once Python receives the user's input, it assigns that input to a variable to make it convenient for you to work with.
- For example, the following program asks the user to enter some text, then displays that message back to the user:

```
message = input("Tell me something, and I will repeat it back to you: ")
print(message)
```

- The input() function takes one argument: the prompt, or instructions, that we want to display to the user so they know what to do. In this example, when Python runs the first line, the user sees the prompt Tell me something, and I will repeat it back to you: . The program waits while the user enters their response and continues after the user presses enter. The response is assigned to the variable message, then print(message) displays the input back to the user.

Numerical Input

- When you use the input() function, Python interprets everything the user enters as a **string**.
- if you try to use the input as a number, you'll get an error ☹
- We can resolve this issue by using the int( ) function, which tells Python to treat the input as a numerical value. The int() function converts a string representation of a number to a numerical representation, as shown here:
  - >>> age = int(input("How old are you? "))
- Type casting: using str( ), int( ), float( ), bool( )
  - *bool( ) returns True for values not equal to zero

Simple examples

```
x = 5

y = 10

print('The value of  x is {} and y is {}'.format(x,y))
```

```
number1 = int( input(‘Enter the first number’) )

number2 = int(input(‘Enter the second number’) )

addition = number1 + number2

print(‘sum of  two numbers {} and {} is: {}’.format(number1,number2,addition))
```

Flow Control: Conditions and Iterations

If Statement

Relational operators

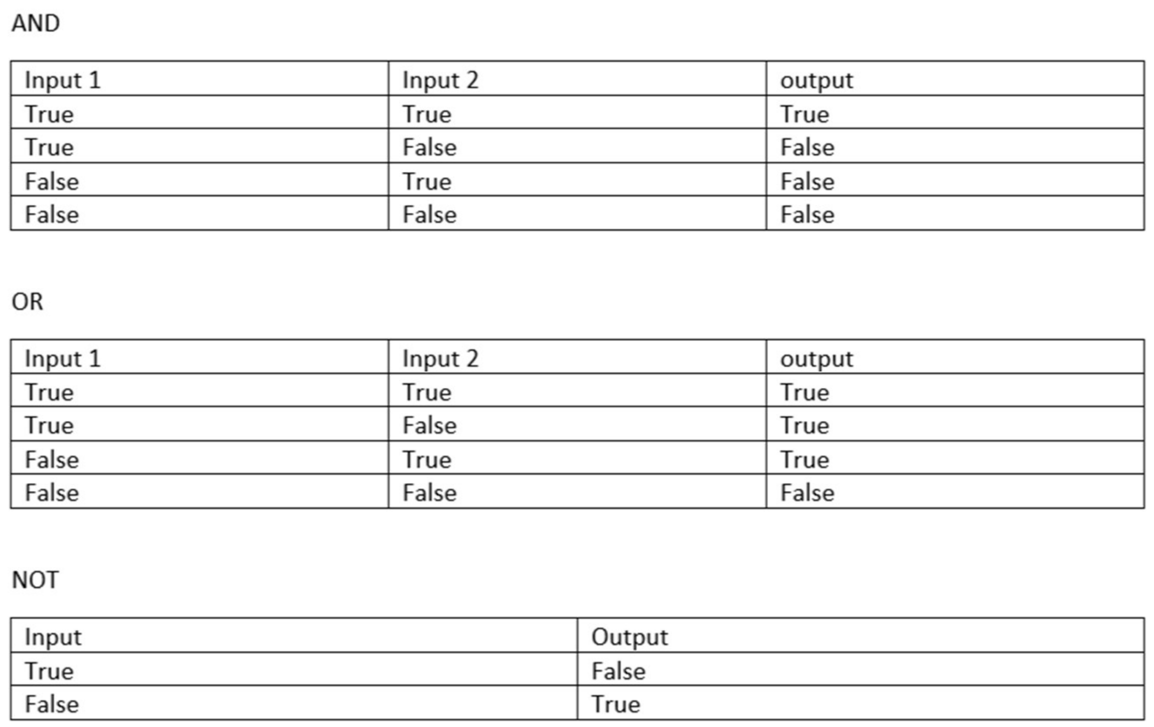
- x == y # x is equal to y
- x != y # x is not equal to y
- x > y # x is greater than y
- x < y # x is less than y
- x >= y # x is greater than or equal to y
- x <= y # x is less than or equal to y

Logical operators

- **and, or, not.**
- For example:
  - x > 0 and x < 10
  - n%2 == 0 or n%3 == 0
  - not (x > y)
- Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as True:

```
>>> 42 and True
True
```

Logical operators



AND

Input 1	Input 2	output
True	True	True
True	False	False
False	True	False
False	False	False

OR

Input 1	Input 2	output
True	True	True
True	False	True
False	True	True
False	False	False

NOT

Input	Output
True	False
False	True

Conditional Execution

- In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the if statement:

```
if <Boolean expression>: # if the expression evaluates to True
    <indented block of code>
```

- Example:

```
if x > 0 :
🟥print('x is positive')
```

- The boolean expression after if is called the condition. If it is true, the indented statement runs. If not, nothing happens.
- There is no limit on the number of statements that can appear in the body (**all should be indented**), but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the pass statement, which does nothing.

```
if x < 0:
    pass # TODO: need to handle negative values!
```

Alternative Execution

- A second form of the if statement is "alternative execution", in which there are two possibilities and the condition determines which one runs. The syntax looks like this:

```
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

- If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays an appropriate message. If the condition is false, the second set of statements runs. Since the condition must be true or false, exactly one of the alternatives will run. The alternatives are called branches, because they are branches in the flow of execution.

Chained Conditionals

- Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

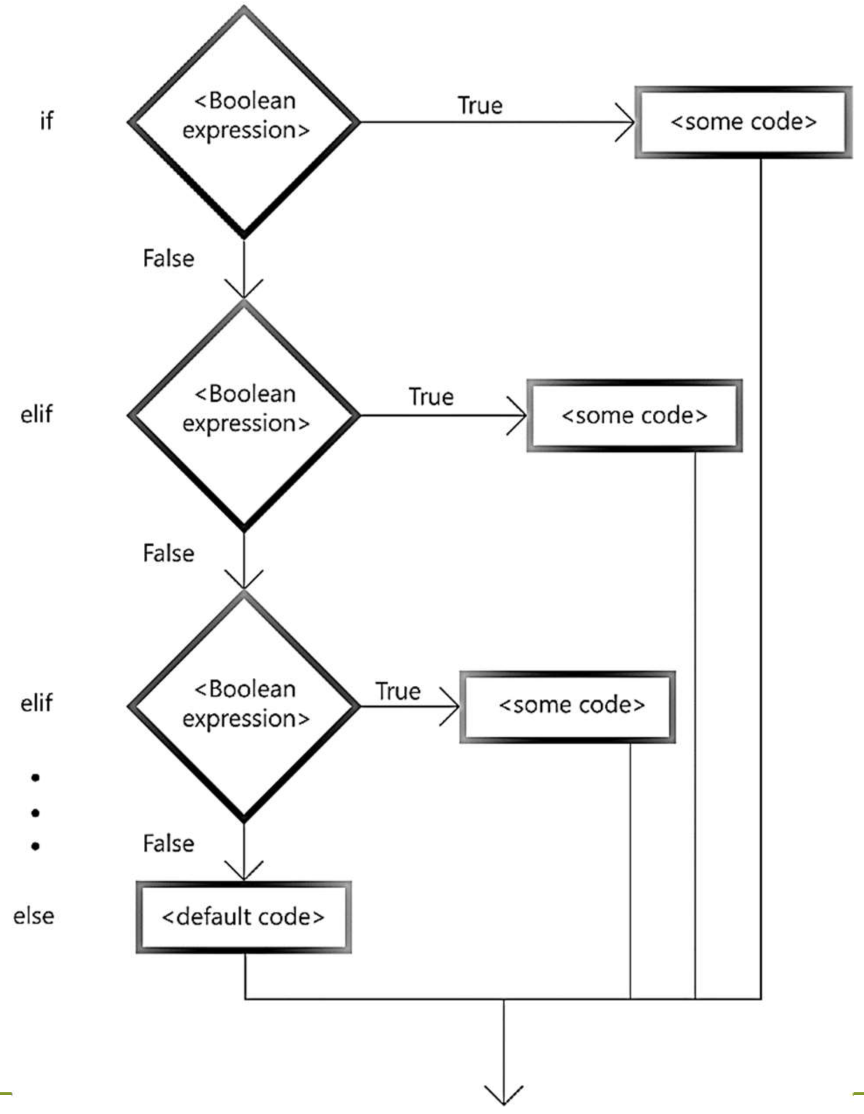
```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

- elif is an abbreviation of "else if ". Again, exactly one branch will run. There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn't have to be one.

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

- Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch runs and the statement ends. Even if more than one condition is true, only the first true branch runs

Flowchart of if/elif/else



Nested Conditionals

- One conditional can also be nested within another:

```
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')
```

Magic 8-Ball children's toy

```
randomAnswer = random.randrange(0, 8) # pick a random number between 0 and 7
if randomAnswer == 0:
	print('It is certain.')
elif randomAnswer == 1:
	print('Absolutely!')
elif randomAnswer == 2:
	print('You may rely on it.')
elif randomAnswer == 3:
	print('Answer is foggy, ask again later.')
elif randomAnswer == 4:
	print('Concentrate and ask again.')
elif randomAnswer == 5:
	print('Unsure at this point, try again.')
elif randomAnswer == 6:
	print('No way, dude!')
else: # must be 7
	print ('No, no, no, no, no.')
```

While Loops

Definition and a silly example

```
while <Boolean expression>: # as long as the expression evaluates to True
    <indented block of code>
```

- Silly example:

```
looping = True
while looping == True:
    answer = input("Please type the letter 'a': ")
    if answer == 'a’:
        looping = False # we're done
    else:
        print("Come on, type an ‘a’!”)
print("Thanks for typing an 'a'")
```

Avoid infinite loops

- If we never have any code that changes the exit condition, then we would create what is called an infinite loop—a loop that runs forever (or until you quit IDLE or shut down your computer).

Serious example

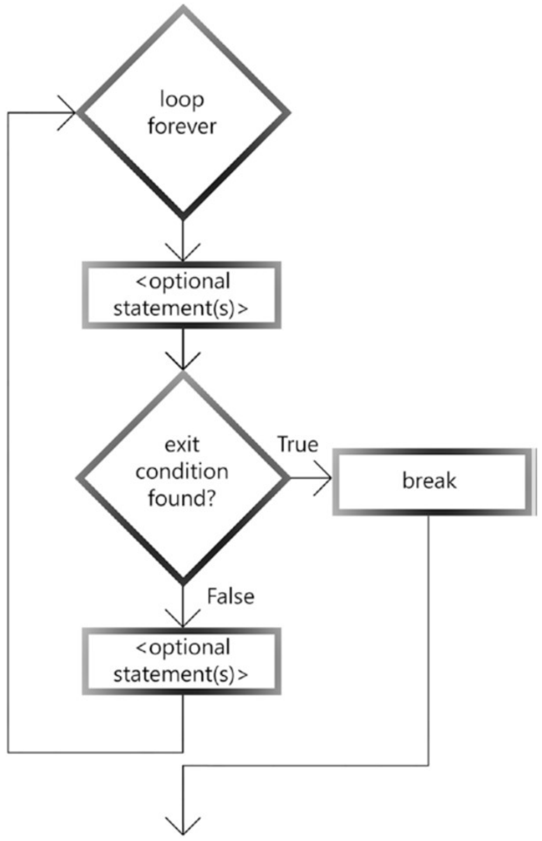
- Let's build a simple program using a loop. The program asks the user for a target number. The goal of the program is to calculate the sum of the numbers from 1 through the target number. For example, if the user enters 4, then we want to calculate 1 + 2 + 3 + 4, and report the answer of 10:

```
#Add up numbers from 1 to a target number
target = input('Enter a target number: ')
target = int(target)
total = 0
nextNumberToAddIn = 1
while nextNumberToAddIn <= target:
    # add in the next value
    total = total + nextNumberToAddIn #add in the next number
    print('Added in:', nextNumberToAddIn, 'Total so far is:', total)
    nextNumberToAddIn = nextNumberToAddIn + 1
print('The sum of the numbers from 1 to', target, 'is:', total)
```

Run till user wants

```
answer = 'y' # start off with the value 'y' to go through the first time
while answer == 'y’:
    usersTarget = input('Enter a target number: ‘)
    target = int(usersTarget)
    total = 0
    nextNumberToAddIn = 1
    while nextNumberToAddIn <= target:
        # add in the next value
        total = total + nextNumberToAddIn  #increment
        nextNumberToAddIn = nextNumberToAddIn + 1
    thisTotal = total
    print('The sum of the numbers 1 to', usersTarget, 'is:', thisTotal)
    answer = input('Do you want to try again (y or n): ')
print('OK Bye')
```

A New Style of Building a Loop: while True, and **break**



```
while True: # loop forever
    line = input("Type anything, type 'done' to exit: ")
    if line == 'done':
        break # transfers control out of the loop
    print('You entered:', line)
print('Finished')
```

For Loops

Definition

- The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects.
- Syntax

```
for val in sequence:
    loop body
```

- Here, val is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

For loop with **range( )**

- We can generate a sequence of numbers using range( ) function. range(10) will generate numbers from 0 to 9 (10 numbers).
- We can also define the start, stop and step size as range(start, stop,step_size). step_size defaults to 1 if not provided.
- Example with one argument

```
for i in range(5):
    print(i, end=", ") # prints: 0, 1, 2, 3, 4,
```

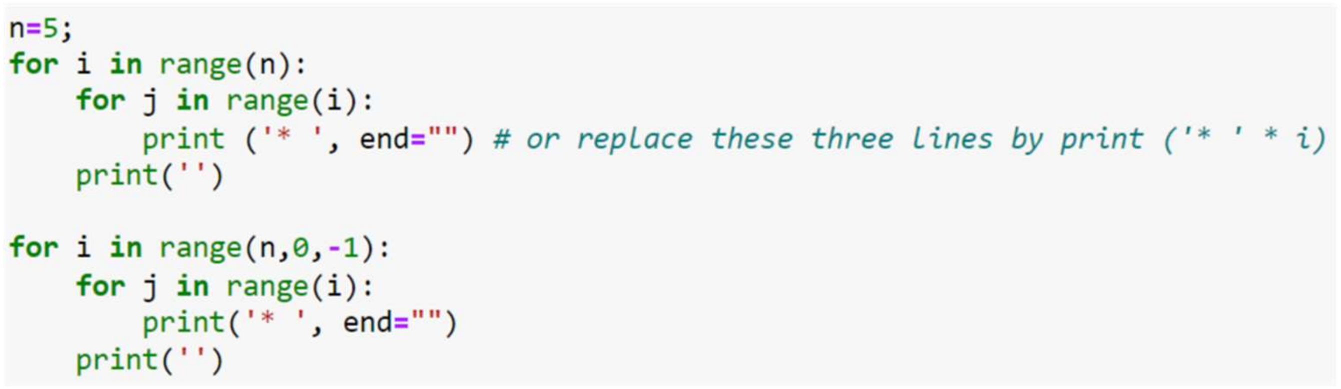
- Example with three arguments

```
for i in range(-1, 5, 2):
    print(i, end=", ") # prints: -1, 1, 3,
```

Examples

- **Write a Python program to construct the following pattern.**

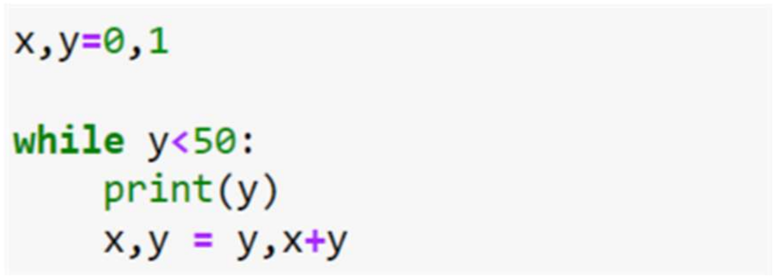
```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```



```python
n=5;
for i in range(n):
    for j in range(i):
        print ('* ', end="") # or replace these three lines by print ('* ' * i)
    print('')

for i in range(n,0,-1):
    for j in range(i):
        print('* ', end="")
    print('')
```

Examples

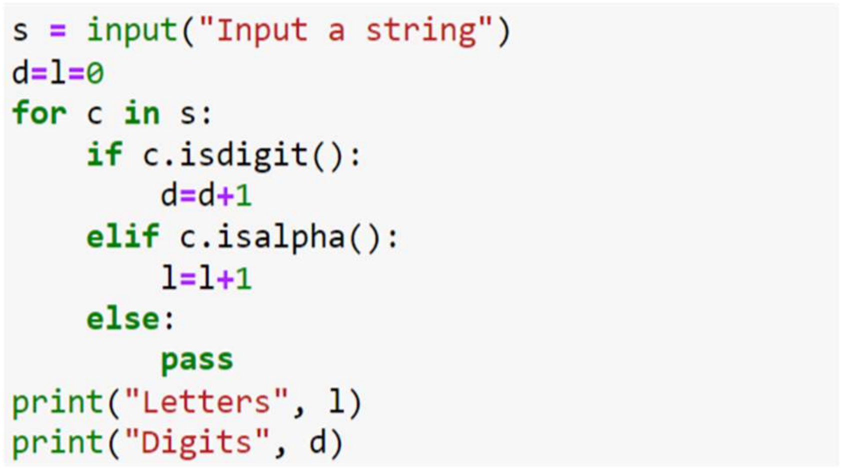
- **Write a Python program to get the Fibonacci series between 0 to 50**



```
x,y=0,1

while y<50:
    print(y)
    x,y = y,x+y
```

Examples

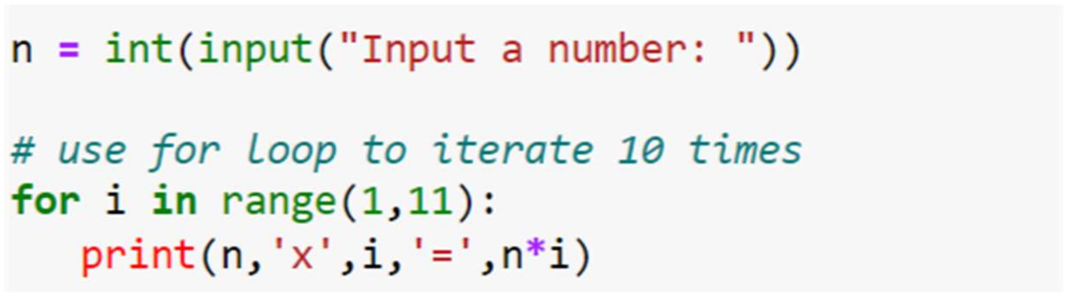
- **Write a Python program that accepts a string and calculate the number of digits and letters.**



```python
s = input("Input a string")
d=l=0
for c in s:
    if c.isdigit():
        d=d+1
    elif c.isalpha():
        l=l+1
    else:
        pass
print("Letters", l)
print("Digits", d)
```

Examples

- **Write a Python program to create the multiplication table (from 1 to 10) of a number.**

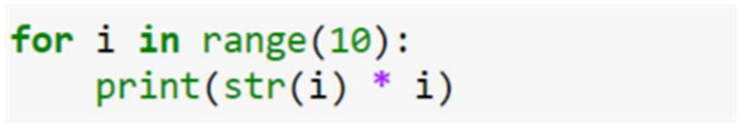


```python
n = int(input("Input a number: "))

# use for loop to iterate 10 times
for i in range(1,11):
   print(n,'x',i,'=',n*i)
```

Examples

- **Write a Python program to construct the following pattern**

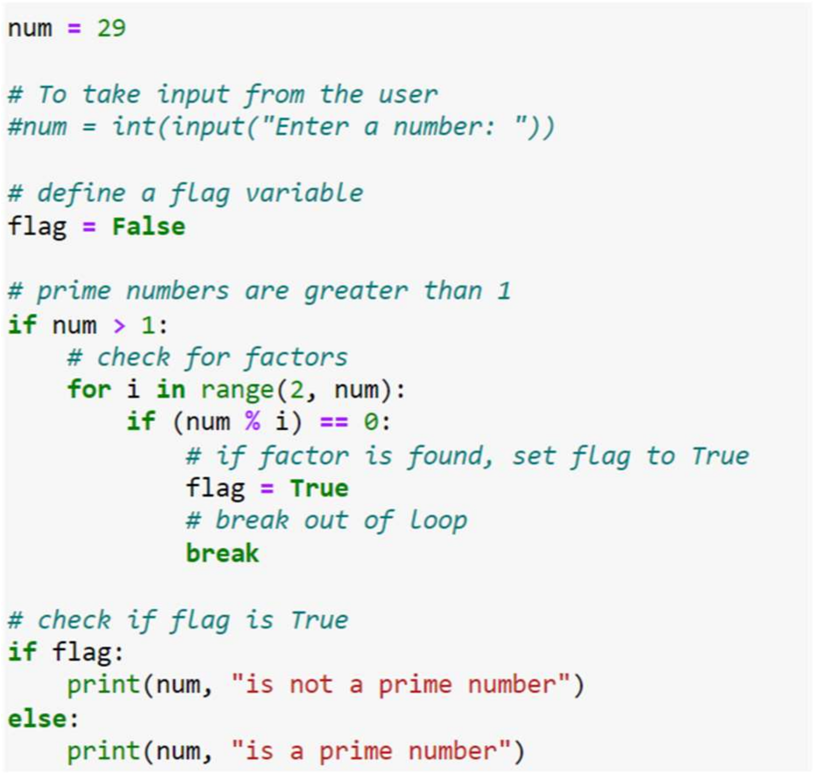
```
1
22
333
4444
55555
666666
7777777
88888888
999999999
```



```
for i in range(10):
    print(str(i) * i)
```

Examples

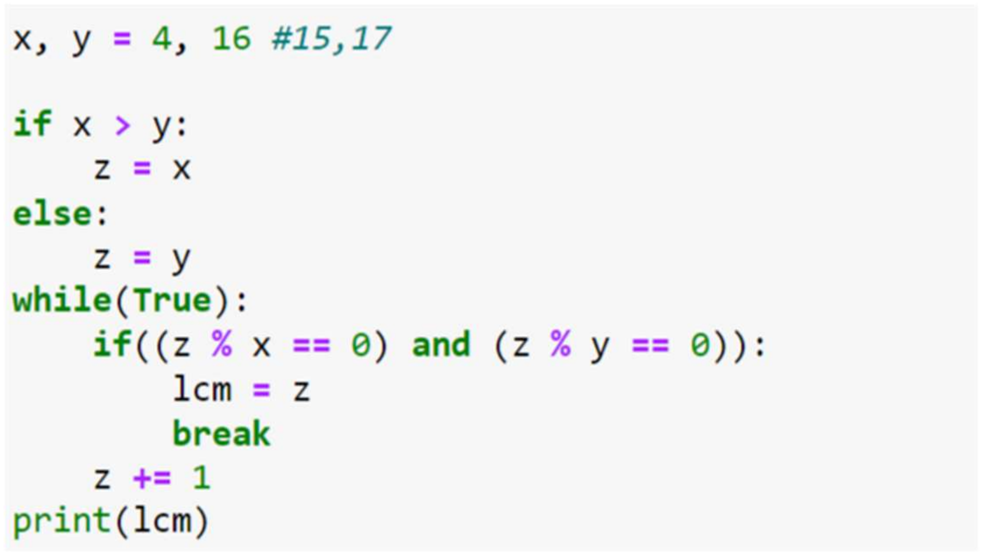
- Write a program to check whether a given number is prime number.



```python
num = 29

# To take input from the user
#num = int(input("Enter a number: "))

# define a flag variable
flag = False

# prime numbers are greater than 1
if num > 1:
    # check for factors
    for i in range(2, num):
        if (num % i) == 0:
            # if factor is found, set flag to True
            flag = True
            # break out of loop
            break

# check if flag is True
if flag:
    print(num, "is not a prime number")
else:
    print(num, "is a prime number")
```

Examples

- Write a program to get the least common multiple of two integers



```python
x, y = 4, 16 #15,17

if x > y:
    z = x
else:
    z = y
while(True):
    if((z % x == 0) and (z % y == 0)):
        lcm = z
        break
    z += 1
print(lcm)
```

List, Dictionary, Tuple, Set in Python

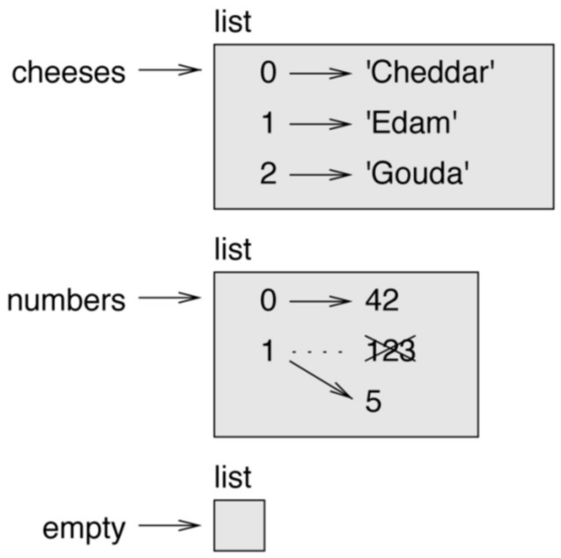
What is a list?

- A list is an ordered sequence of values.
  - the values in a list can be **any** type.
  - the values in a list are called elements or items.
- Examples of lists:
  - Name of students registered in a course
    - [‘Maryam’, ‘Mina’, ‘Ali’, ‘Reza’, ‘Narges’]
  - Students’ grades
    - [20, 19, 15, 18, 13]

How to create lists?

- Some examples:
  - empty = [ ]
  - numbers = [42, 123]
  - cheeses = [‘Cheddar’, ‘Edam’, ‘Gouda’]
  - mylist = [1, 2, 3, 4, 4, 4, 5, 5, ‘a’, ‘b’]
  - nested = [‘spam’, 2.0, 5, [10, 20]]
- A list that contains no elements is called an empty list.
- A list within another list is nested.
- Singleton: a list that contains a single element.

Accessing the elements of a list



- The syntax for accessing the elements of a list is by using the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0.
- Lists are **mutable**.
- >>> cheeses[0]

  'Cheddar'
- >>> numbers[1] = 5
- >>> numbers

  [42, 5]

Accessing the elements of a list

- Python has a special syntax for accessing the last element in a list. By asking for the item at index -1, Python always returns the last item in the list.
- This syntax is quite useful, because you'll often want to access the last items in a list without knowing exactly how long the list is. This convention extends to other negative index values as well.

```
• >>> numbers = [4, 43, 129]
• >>> numbers[-1]
  129
• >>> numbers[-2]
  43
```

Changing, Adding, and Removing Elements

Most lists you create will be **dynamic**, meaning you'll build a list and then add and remove elements from it as your program runs its course.

Modifying Elements in a List

- Modifying Elements in a List

- >>> numbers = [3, 4, 10]
- >>> numbers[0] = 12

Adding Elements to a List

- Appending elements to the end of a list using append( ) method.
  - >>>numbers = [3, 4, 10]
  - >>>numbers.append(23)

    numbers = [3, 4, 10, 23]
- Inserting elements into a list
  - >>> numbers.insert(0, 78)

    numbers = [78, 3, 4, 10, 23]
  - opens a space at position 0 and stores the value 78 at that location.

Changing, Adding, and Removing Elements

Removing Elements from a List

- We know the position of the item we want to remove:
  - Use del statement

```
>>> numbers = [3, 4, 10]
>>> del numbers[1]
```

- We want to use the value of an item after removing it from a list:
  - Use pop( ) method. The pop( ) method removes the last item in a list, but it lets you work with that item after removing it.

```
>>> numbers = [3, 4, 10]
>>> last_number = numbers.pop( )
        numbers = [3, 4]
        last_number = 10
```

  - You can use pop( ) to remove an item from any position in a list by including the index of the item you want to remove in parentheses.

```
>>> numbers.pop(0)
       numbers = [4]
```

Changing, Adding, and Removing Elements

Removing an Item by Value

- We don’t know the position of the value we want to remove from a list. If we only know the value of the item we want to remove, we can use the remove( ) method.

```
>>> numbers = [3, 4, 10]

>>> numbers.remove(4)
```

Lists and Strings

From string to list

- string to a list of characters:

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm’]
```

- string into words:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
['spam', 'spam', 'spam']
```

List to string

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s
'pining for the fjords'
```

Useful methods/operators for manipulation

- Finding the length of a list: len ( )

```
>>> len(numbers)
```

- Sorting a list permanently with the sort( ) method

```
>>> numbers.sort( )
```

- Sorting a list temporarily with the sorted() function

```
>>> sorted(numbers)
```

- Extending a list by more than one element
  - `>>> numbers.extend( [3,4] )`
- Clear( ), index(item), count(item),sum(list),…

- The + operator concatenates lists
- The * operator repeats a list a given number of times
- **List Slices:**

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
    ['b', 'c']
>>> t[:4]
    ['a', 'b', 'c', 'd']
>>> t[3:]
    ['d', 'e', 'f']
>>> t[:]
    ['a', 'b', 'c', 'd', 'e', 'f']
```

Possible mistakes!

Aliasing

- If a refers to an object and you assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
    True
```

- If the aliased object is mutable, changes made with one alias affect the other:

```
>>> b[0] = 42
>>> a
    [42, 2, 3]
```

- Although this behavior can be useful, it is error-prone.

Possible Mistakes!

Avoiding index errors

- An index error means Python can't find an item at the index you requested. If an index error occurs in your program, try adjusting the index you're asking for by one.

```
>>>numbers = [3, 4, 10]
>>>numbers[3]
```

Possible Mistakes!

- It is important to distinguish between operations that modify lists and operations that create new lists.

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
    [1, 2, 3]
>>> t2
     None
```

- To add an element, what we shouldn't do:
  - t.append([x])
  - t = t.append(x)
  - t + [x]
  - t = t + x

Possible Mistakes!

- **Make copies to avoid aliasing!**
- If you want to use a method like sort that modifies the argument, but you need to keep the original list as well, you can make a copy:

```
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>> t2.sort()
>>> t
    [3, 1, 2]
>>> t2
    [1, 2, 3]
```

- In this example you could also use the built-in function sorted, which returns a new, sorted list and leaves the original alone

Hands on codes!

- adding up the elements from all of a nested lists:

```
t = [ [ 1, 2 ] , [ 3 ], [ 4, 5, 6 ] ]
total = 0
for nested in t:
    total += sum(nested)
```

- Taking a list of numbers and returning the cumulative sum; that is, a new list where the ith element is the sum of the first i+1 elements from the original list.

```
total = 0
res = []
t = [ 1, 2, 3 ]
for x in t:
   total += x
   res.append(total)
```

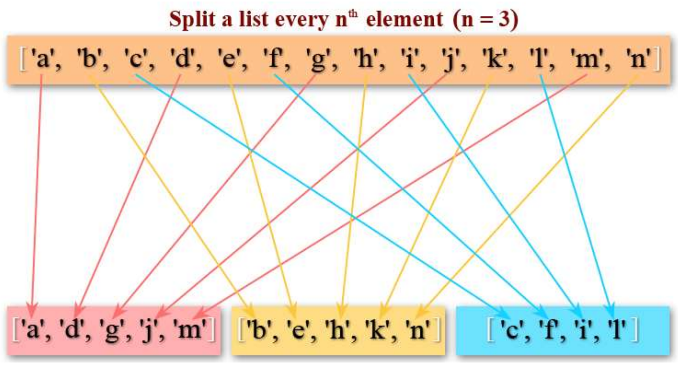
- Check if there is any element in a given list that appears more than once.

```
s.sort()
check = False
# check for adjacent elements that are equal
for i in range(len(s)-1):
   if s[i] == s[i+1]:
         check = True
```

- Select an item randomly from a list.

```
import random
t = [ 2, 45, 21, 56 ]
print(random.choice(t))
```

- split a list every n-th element.



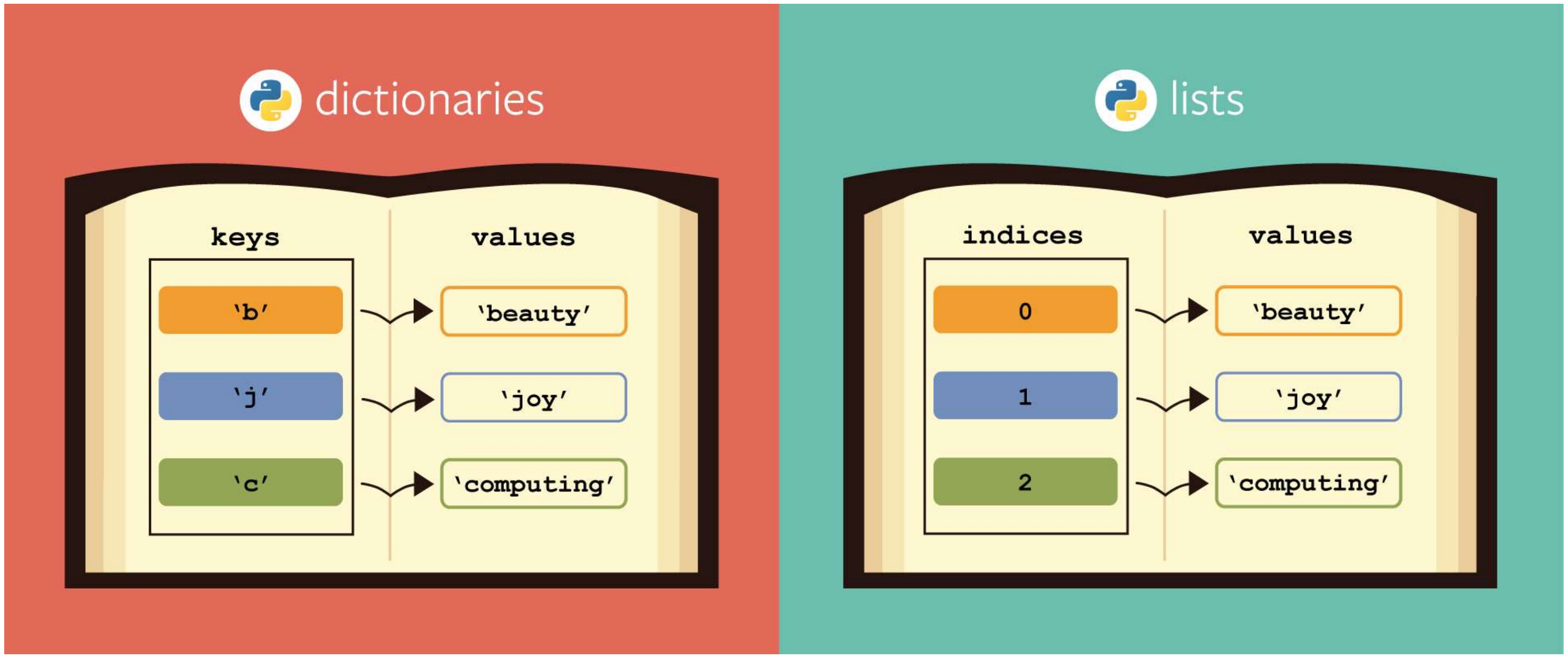
Split a list every nth element (n = 3)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n']
['a', 'd', 'g', 'j', 'm']
['b', 'e', 'h', 'k', 'n']
['c', 'f', 'i', 'l']

```
C = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n']
A = [C[i::step] for i in range(step)]
```

Dictionary

- A dictionary in Python is a collection of key-value pairs. A dictionary is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type. A dictionary contains a collection of indices, which are called **keys**, and a collection of **values**. Each key is associated with a single value.
- dictionary represents a mapping from keys to values, so you can also say that each key "maps to" a value.

Dictionary


dictionaries
keys
values
'b'
'beauty'
'j'
'joy'
'c'
'computing'
lists
indices
values
0
'beauty'
1
'joy'
2
'computing'

Creating dictionary

- The function dict( ) creates a new dictionary with no items.

```
>>> eng2fr = dict()
>>> eng2fr
{}
```

- The squiggly brackets, { }, represent an empty dictionary.

```
>>> eng2fr [ ‘one’ ] = ‘un’
>>> eng2fr
{‘one’: ‘un’}
```

- Or Add multiple entries:

```
>>> eng2fr = {‘one’: ‘un’, ‘two’: ‘duex’, ‘three’: ‘trois’}
```

- Look up values:

```
>>> eng2fr['two']
‘duex’
```

- The len function works on dictionaries; it returns the number of key-value pairs
- The in operator works on dictionaries, too; it tells you whether something appears as a key in the dictionary

```
>>> 'one' in eng2fr
True
```

- To see whether something appears as a value in a dictionary, use the method values, which returns a collection of values, and then use the in operator:

```
>>> vals = eng2fr.values()
>>> 'un' in vals
True
```

Accessing the dictionary

- Dictionaries have a method called get that takes a key and a default value. If the key appears in the dictionary, get returns the corresponding value; otherwise it returns the default value. For example:

```
>>> eng2fr.get('four', 0)
0
```

- Traversing the keys of the dictionary:

```
>>>for c in eng2fr:
        print(c, eng2fr[c])
```

- The keys are in no particular order. To traverse the keys in sorted order, use the built-in function sorted:

```
>>>for key in sorted(eng2fr):
        print(key, eng2fr[key])
```

Look-up and reverse look-up

- Given a dictionary d and a key k, it is easy to find the corresponding value v = d[k].
- This operation is called a lookup.

- But what if you have v and you want to find k?

```
>>>key = none
    for k in d:
      if d[k] == v:
        key = k
```

Dictionary & list

- Lists can be values in a dictionary, but they cannot be keys.
- Here's what happens if you try:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

- A dictionary is implemented using a hashtable and that means that the keys have to be hashable. A hash is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs.
- This system works fine if the keys are immutable. But if the keys are mutable, like lists, bad things happen. For example, when you create a key-value pair, Python hashes the key and stores it in the corresponding location. If you modify the key and then hash it again, it would go to a different location. In that case you might have two entries for the same key, or you might not be able to find a key. Either way, the dictionary wouldn't work correctly.
- That's why keys have to be hashable, and why mutable types like lists aren't. The simplest way to get around this limitation is to use **tuples**.

Tuples

- A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that **tuples are immutable**.
- it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e’)
```

- To create a tuple with a single element, you have to include a final comma:

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

- A value in parentheses is not a tuple:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

Creating Tuples

- Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>> t
( )
```

- If the argument is a sequence (string, list), the result is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

- You can use tuple assignment in a for loop to traverse a list of tuples:

```
>>>t = [('a', 0), ('b', 1), ('c', 2)]
        for letter, number in t:
            print(number, letter)
```

- Traverse the elements of a sequence and their indices, you can use the built-in function enumerate:

```
>>>for index, element in enumerate('abc'):
        print(index, element)
```

Dictionary & Tuple

- Dictionaries have a method called items that returns a sequence of tuples, where each tuple is a key-value pair:

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

- You can use it in a for loop like this:

```
>>> for key, value in d.items():
        print(key, value)
```

- You can use a list of tuples to initialize a new dictionary:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

Sets

- A set is a collection which is unordered, unindexed and do not allow duplicate values.

```
set1 = {"apple", "banana", "cherry"}

set2 = {1, 5, 7, 9, 3}

set3 = {True, False, False}

set4 = {"abc", 34, True, 40, "male"}
```

Creating Sets

- Creating an empty set is a bit tricky.
- Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements, we use the set() function without any argument.
  - initialize a dictionary with {}

    a = {}

  - initialize a with set( )

    a = set( )

- initialize my_set

  my_set = {1, 3}
- What happens if we ask: my_set[0]
  - We will get an error

    TypeError: 'set' object does not support indexing
- Add an element:

  my_set.add(2)
- Add multiple elements

  my_set.update([2, 3, 4])

Manipulating Sets

- Removing elements from a set:

  discard( ) & remove( )

- Random elimination:

  pop( )

- Remove all elements:

  clear ( )

- Set operations:

  union, intersection, difference

Hands on code!

- Write a Python program to get unique values from a list.

```
my_list = [10, 20, 30, 40, 20, 50, 60, 40]
print("Original List : ", my_list)
my_set = set(my_list)
my_new_list = list(my_set)
print("List of unique numbers : ", my_new_list)
```

- Write a Python script to merge two Python dictionaries.

```
d1 = {'a': 100, 'b': 200}
d2 = {'x': 300, 'y': 200}
d = d1.copy()
d.update(d2)
```

- Write a Python script to print a dictionary where the keys are numbers between 1 and 15 (both included) and the values are square of keys.

```
d=dict()
for x in range(1,16):
    d[x]=x**2
```

- Write a Python program to sum all the items in a dictionary.

```
my_dict = { 'data1':100 , 'data2':-54 , 'data3':247 }
print( sum ( my_dict.values( ) ) )
```