Scientific Programming with Python

Lecture 2: Imperative Programming

UNIVERSITY of HOUSTON
DIVISION OF RESEARCH
HEWLETT PACKARD ENTERPRISE DATA SCIENCE INSTITUTE

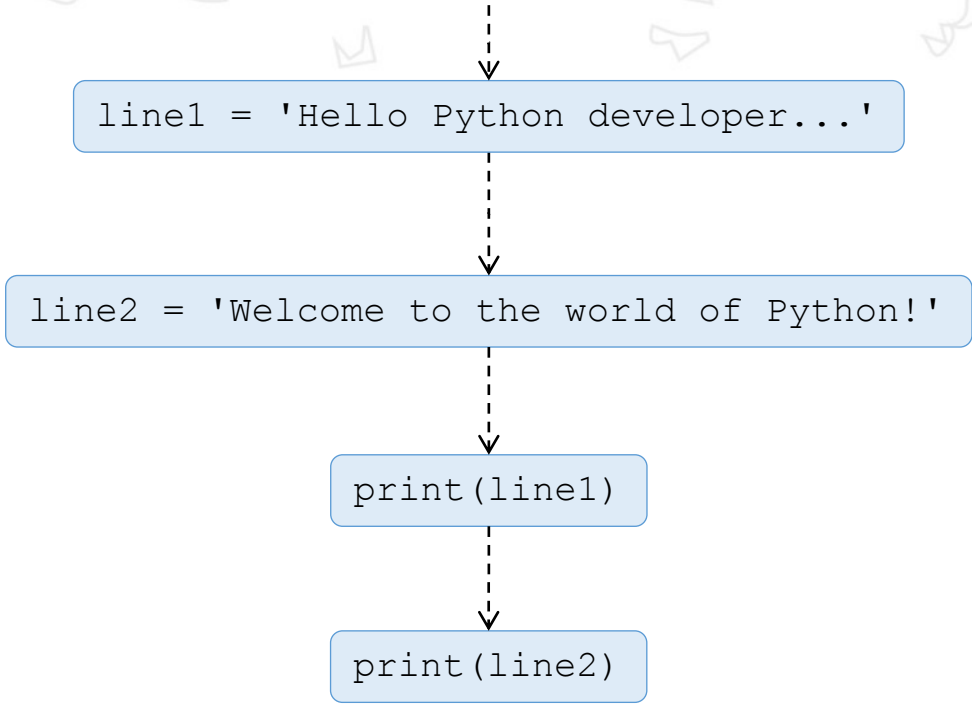
Lecture 2: Imperative Programming

- Python Programs
- Interactive Input/Output
- One-Way and Two-Way `if` Statements
- `for` Loops
- While loops
- User-Defined Functions
- Exception handling

Python program

A Python program is a sequence of Python statements

- Stored in a text file called a Python module
- Executed using an IDE or “from the command line”




```
line1 = 'Hello Python developer...'
line2 = 'Welcome to the world of Python!'
print(line1)
print(line2)
```

hello.py

```
$ python hello.py
Hello Python developer…
```

Built-in function `input()`

Function `input()` requests and reads input from the user interactively

- It's (optional) input argument is the request message
- Typically used on the right side of an assignment statement

When executed:

1. The input request message is printed
2. The user enters the input
3. The *string* typed by the user is assigned to the variable on the left side of the assignment statement

```
>>> name = input('Enter your name: ')
Enter your name: Michael
>>> name
'Michael'
>>> ========= RESTART ============
>>>
Enter your first name:
```

```
first = input('Enter your first name: ')
last = input('Enter your last name: ')
line1 = 'Hello' + first + '' + last + '…'
print(line1)
print('Welcome to the world of Python!')
```

input.py

Built-in function `eval()`

Function `input()` evaluates anything the user enters as a string

What if we want the user to interactively enter non-string input such as a number?

- Solution 1: Use type conversion
- Solution 2: Use function `eval()`
  - Takes a string as input and evaluates it as a Python expression

```
>>> age = input('Enter your age: ')
Enter your age: 18
>>> age
'18'
>>> int(age)
18
>>> eval('18')
18
>>> eval('age')
'18'
>>> eval('[2,3+5]')
[2, 8]
>>> eval('x')
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in
<module>
    eval('x')
  File "<string>", line 1, in
<module>
NameError: name 'x' is not defined
>>>
```

Exercise

Write a program that:

1. Requests the user's name
2. Requests the user's age
3. Computes the user's age one year from now and prints the message shown

```
>>>
Enter your name: Marie
Enter your age: 17
Marie, you will be 18 next year!
```

```
name = input('Enter your name: ')
age = int(input('Enter your age: '))
line = name + ', you will be ' + str(age+1) + ' next year!'
print(line)
```

Exercise

Write a program that:

1. Requests the user's name
2. Requests the user's age
3. Prints a message saying whether the user is **eligible to vote or not**

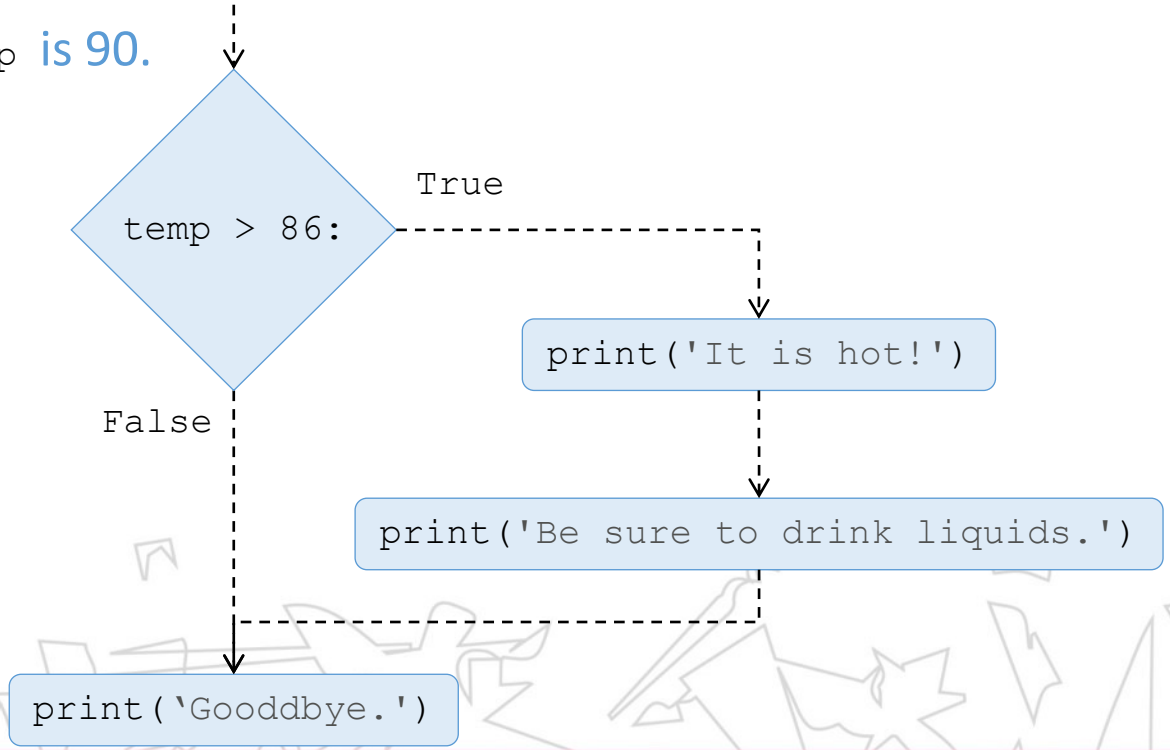
Need a way to execute a Python statement if a condition is true

One-way if statement

```
if <condition>:
    <indented code block>
<non-indented statement>
```

```
if temp > 86:
    print('It is hot!')
    print('Be sure to drink liquids.')
print('Goodbye.')
```

Assume the value of `temp` is 90.



Examples

Write corresponding if statements:

a) If `age` is greater than 62 then print `'You can get Social Security benefits'`

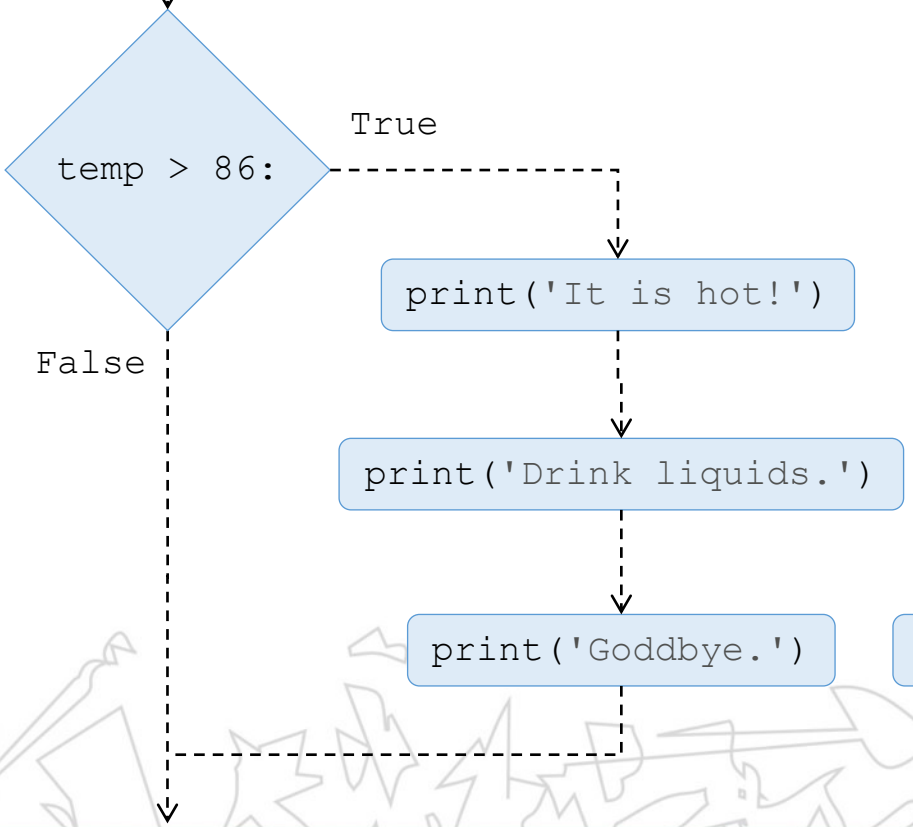
b) If string `'large bonuses'` appears in string `report` then print `'Vacation time!'`

c) If `hits` is greater than 10 and `shield` is 0 then print `"You're dead..."`

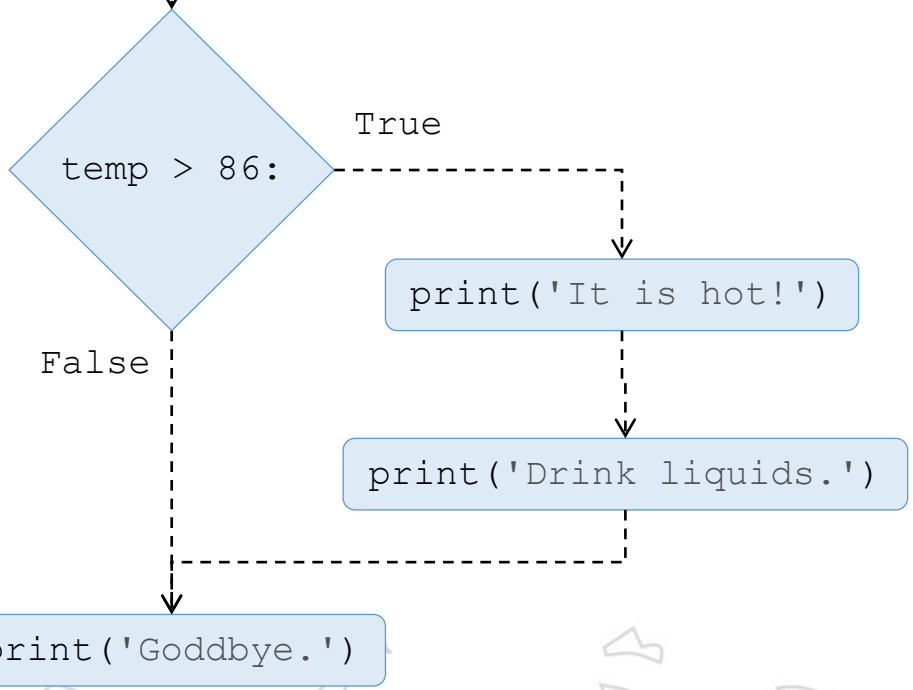
```
>>> hits = 12
>>> shield = 0
>>> if hits > 10 and shield == 0:
        print("You're dead...")


You're dead...
>>> hits, shield = 12, 2
>>> if hits > 10 and shield == 0:
        print("You're dead...")


>>>
```

Indentation is critical

```
if temp > 86:
    print('It is hot!')
    print('Drink liquids.')
    print('Goodbye.')
```




```
if temp > 86:
    print('It is hot!')
    print('Drink liquids.')
print('Goodbye.')
```

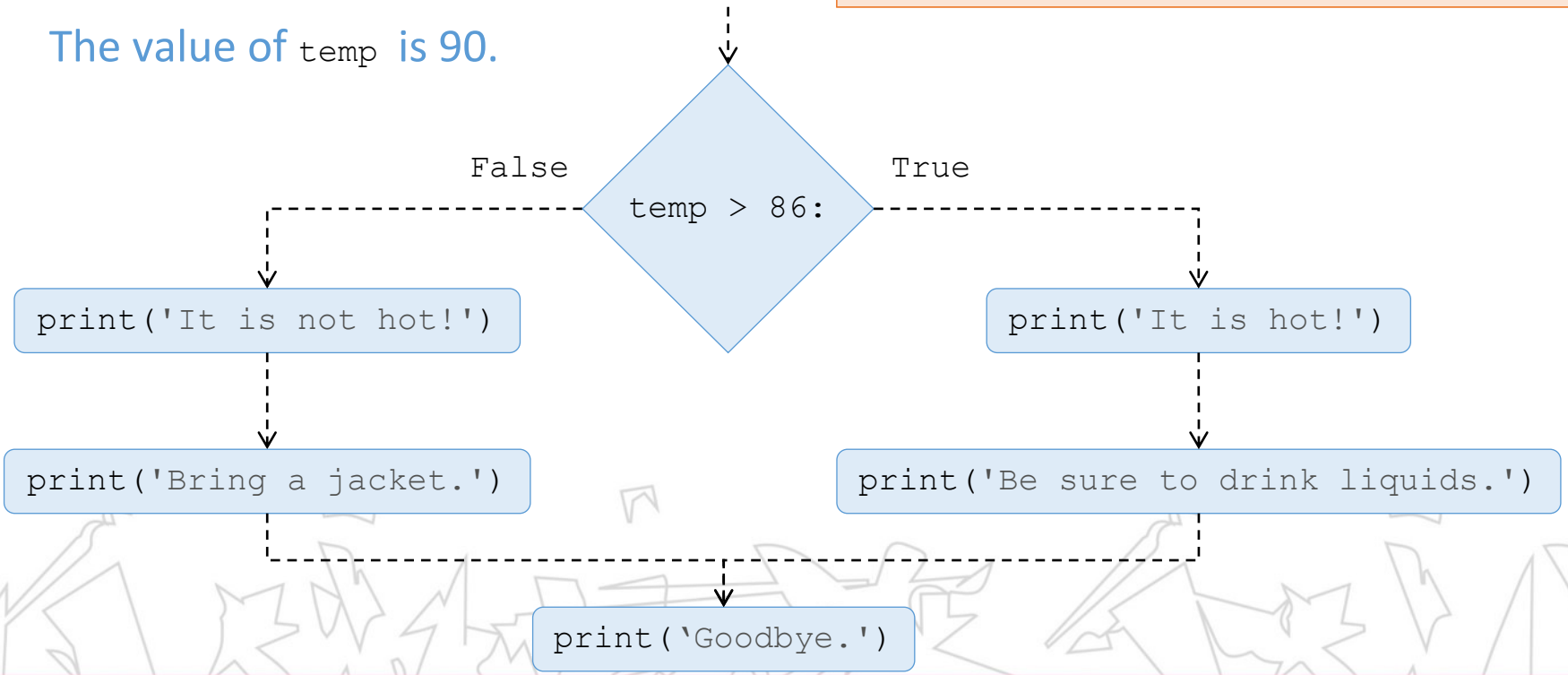


Two-way if statement

```
if <condition>:
    <indented code block 1>
else:
    <indented code block 2>
<non-indented statement>
```

```
if temp > 86:
    print('It is hot!')
    print('Be sure to drink liquids.')
else:
    print('It is not hot.')
    print('Bring a jacket.')
print('Goodbye.')
```

The value of `temp` is 90.



Multi-way if statement

```
if <condition>:
    <indented code block 1>
elif <condition>:
    <indented code block 2>

. . .

elif <condition>:
    <indented code block n-2>
elif <condition>:
    <indented code block n-1>
else:
    <indented code block n>
<non-indented statement>
```

```
print(”Letter Grades”)

score =float (input (“enter raw score”))

prefix=‘Your letter grade is’
if score < 60 :
        print(prefix, ‘F’)
elif   60 <= score  < 70:
        print(prefix, ‘D’)
elif   70 <= score  < 80:
        print(prefix, ‘C’)
elif   80 <= score  < 90:
        print(prefix, ‘B’)
else:
        print(prefix, ‘A’)

print('Goodbye.')
```

Exercise

Write a program that:

1) Requests the user's name
2) Requests the user's age
3) Prints a message saying whether the user is eligible to vote or not

```
>>>
Enter your name: Marie
Enter your age: 17
Marie, you can't vote.
>>>
============RESTART================
>>>
Enter your name: Marie
Enter your age: 18
Marie, you can vote.
>>>
```

```
name = input('Enter your name: ')
age = eval(input('Enter your age: '))
if age < 18:
    print(name + ", you can't vote.")
else:
    print(name + ", you can vote.")
```

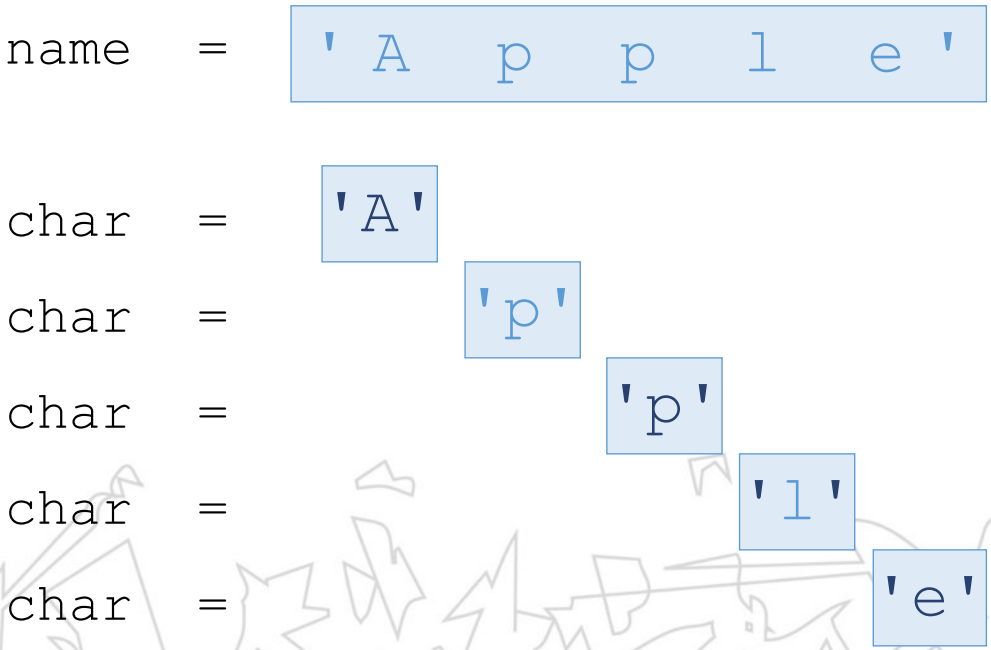
Execution control structures

- The one-way and two-way if statements are examples of execution control structures
- Execution control structures are programming language statements that control which statements are executed, i.e., the execution flow of the program
- The one-way, two-way and multi-way if statements are, more specifically, conditional structures
- Iteration structures are execution control structures that enable the repetitive execution of a statement or a block of statements
  - The for loop statement is an iteration structure that executes a block of code for every item of a sequence

for loop

Executes a block of code for every item of a sequence

- If sequence is a string, items are its characters (single-character strings)




```
>>> name = 'Apple'
>>> for char in name:
        print(char)


A
p
p
l
```

for loop

Executes a code block for every item of a sequence

- Sequence can be a string, a list, …
- Block of code must be indented

```
for <variable> in <sequence>:
    <indented code block >
<non-indented code block>
```

```
for word in ['stop', 'desktop', 'post', 'top']:
    if 'top' in word:
        print(word)
print('Done.')
```

word = 'stop'

word = 'desktop'

word = 'post'

word = 'top'

```
>>>
stop
desktop
top
Done.
```

Exercise

Write a “spelling” program that:

1) Requests a word from the user
2) Prints the characters in the word from left to right, one per line

```
name = input('Enter a word: ')
print('The word spelled out: ')

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

```
============RESTART================
>>>
Enter a word: omnipotent
The word spelled out:
o
m
n
i
p
o
t
e
n
t
>>>
```

Built-in function `range()`

Function range() is used to iterate over a sequence of numbers in a specified range

- represents an immutable sequence of numbers
- takes the same small amount of memory, compared to lists
  - does not actually store the individual values representing the number sequence
  - instead, range function as iterators and calculate the sequence values on the fly

- To iterate over the n numbers, starting at 0. 0, 1, 2, …, n-1

```
for i in range(n):
```

- To iterate over the n numbers, starting at i. i, i+1, i+2, …, n-1

```
for i in range(i, n):
```

- To iterate over the n numbers i, i+c, i+2c, i+3c, …, n-1

```
for i in range(i, n, c):
```

```
>>> for i in range(2, 16, 10):
        print(i)


2
12
>>>
```

Exercise

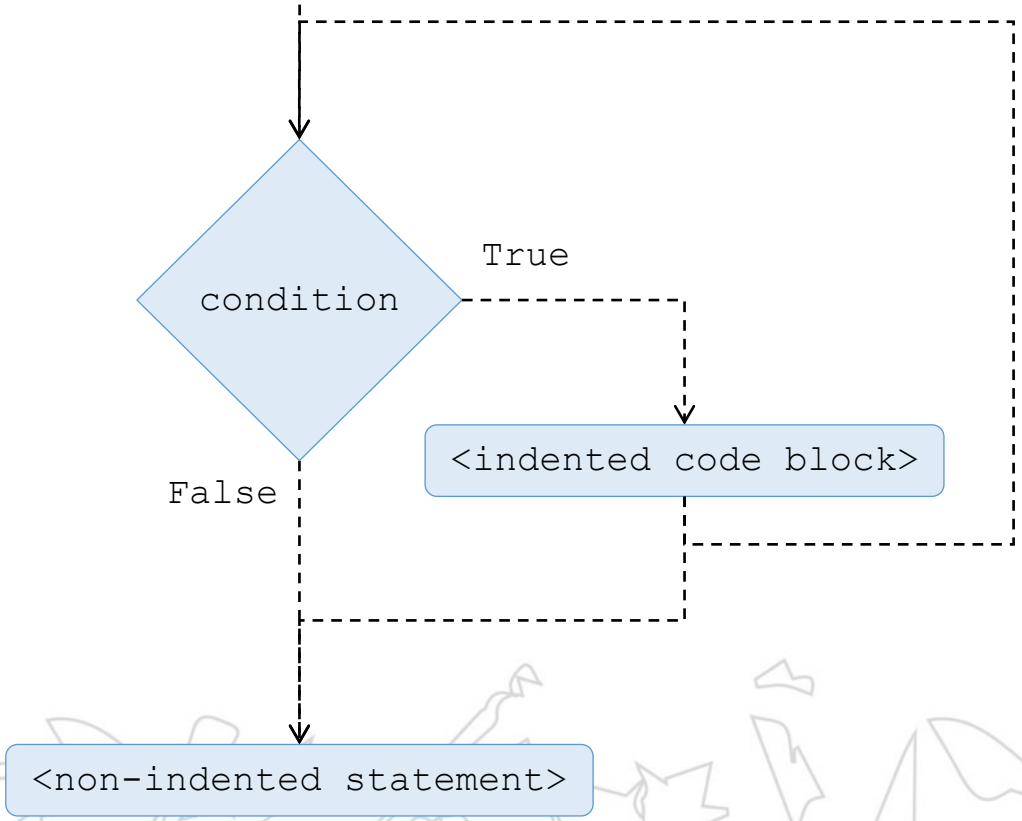
Write for loops that will print the following sequences:

a) 0, 1, 2, 3, 4, 5, 6, 7, 8 , 9, 10
b) 1, 2, 3, 4, 5, 6, 7, 8, 9
c) 0, 2, 4, 6, 8
d) 1, 3, 5, 7, 9
e) 20, 30, 40, 50, 60

while loop

```
if <condition>:
    <indented code block>
<non-indented statement>
```

```
while <condition>:
    <indented code block>
<non-indented statement>
```

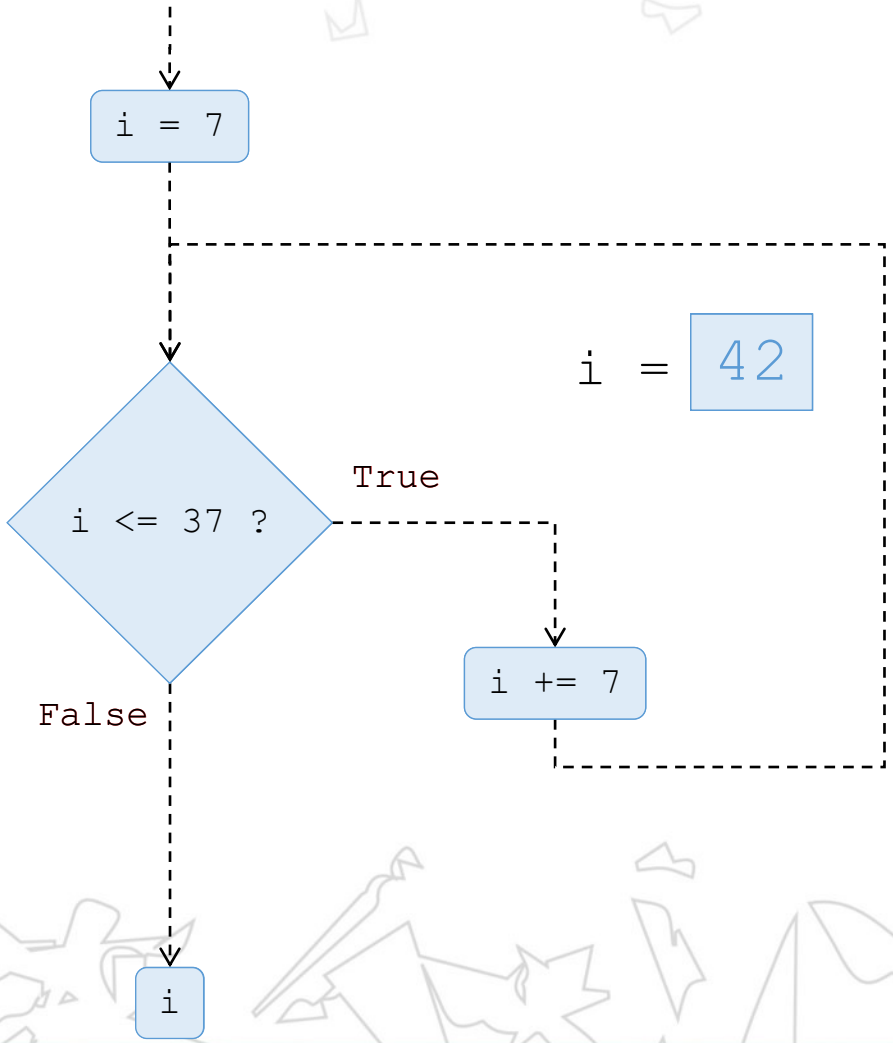


while loop

Example: compute the smallest multiple of 7 greater than 37.

Idea: generate multiples of 7 until we get a number greater than 37

```
>>> i = 7
>>> while i <= 37:
        i = i+ 7

>>> i
42
```



Defining new functions

A few built-in functions we have seen:

- `abs()`, `max()`, `len()`, `sum()`, `print()`

New functions can be defined using `def`

`def`: function definition keyword

`f`: name of function

`x`: variable name for input argument

```
def f(x):
    res = x**2 + 10
    return res
```

`return`: specifies function output

```
>>> abs(-9)
9
>>> max(2, 4)
4
>>> lst = [2,3,4,5]
>>> len(lst)
4
>>> sum(lst)
14
>>> print()

>>> def f(x):
        res = 2*x + 10
        return x**2 + 10

>>> f(1)
11
>>> f(3)
19
>>> f(0)
10
```

print() versus return

```
def f(x):
    res = x**2 + 10
    return res
```

```
>>> f(2)
14
>>> 2*f(2)
28
```

Function returns value of `res` which can then be used in an expression

```
def f(x):
    res = x**2 + 10
    print(res)
```

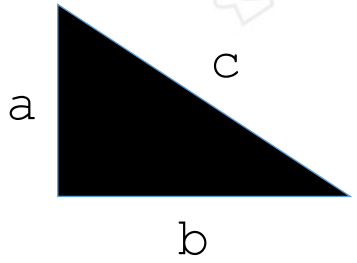
```
>>> f(2)
14
>>> 2*f(2)
14
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in
<module>
    2*f(2)
TypeError: unsupported operand
type(s) for *: 'int' and
'NoneType'
```

Function prints value of `res` but does not return anything

Defining new functions

The general format of a function definition is

```
def <function name> (<0 or more variables>):
    <indented function body>
```



Let's develop function `hyp()` that:

- Takes two numbers as input (side lengths a and b of above right triangle)
- Returns the length of the hypotenuse c

```
>>> hyp(3,4)
5.0
>>>
```

```
import math
def hyp(a, b):
    res = math.sqrt(a**2 + b**2)
    return res
```

Anonymous functions - Lambda expressions

Python supports anonymous functions using lambda expressions

Example with a named function:

```
def hyp(a, b):
    res = a**2 + b**2
    res = res **0.5
    return res
```

C=hyp(a,b) ## usage

Example with an anonymous f function via lambda:

```
hyp= lambda a,b : (a**2 + b**2)**0.5
```

C= hyp(a,b) ## usage

Limitation:

The executable body of the lambda must be a one line expression and can't be a statement

Exercise

Write function `hello()` that:

- takes a name (i.e., a string) as input
- prints a personalized welcome message

Note that the function does not return anything

```
>>> hello('Julie')
Welcome, Julie, to the world of Python.
>>>
```

```
def hello(name):
    line = 'Welcome, ' + name + ', to the world of Python.'
    print(line)
```

Exercise

Write function **`rng()`** that:

- takes a list of numbers as input
- returns the range of the numbers in the list

The range is the difference between the largest and smallest number in the list

```
>>> rng([4, 0, 1, -2])
6
>>>
```

```
def rng(lst):
    res = max(lst) - min(lst)
    return res
```

Comments and docstrings

Python programs should be documented

- So the developer who writes/maintains the code understands it
- So the user knows what the program does

Comments

```
def f(x):
    res = x**2 + 10    # compute result
    return res         # and return it
```

Docstring

```
def f(x):
    'returns x**2 + 10'
    res = x**2 + 10    # compute result
    return res         # and return it
```

```
>>> help(f)
Help on function f in module
__main__:

f(x)

>>> def f(x):
        'returns x**2 + 10'
        res = x**2 + 10
        return res

>>> help(f)
Help on function f in module
__main__:

f(x)
    returns x**2 + 10

>>>
```

Exercise

Write function `negative()` that:

- takes a list of numbers as input
- returns the index of the first negative number in the list or -1 if there is no negative number in the list

```
>>> lst = [3, 1, -7, -4, 9, -2]
>>> negative(lst)
2
>>> negative([1, 2, 3])
-1
```

```
def negative(lst):

    for i in range(len(lst)):
        if lst[i] < 0:
            return i

    return -1
```

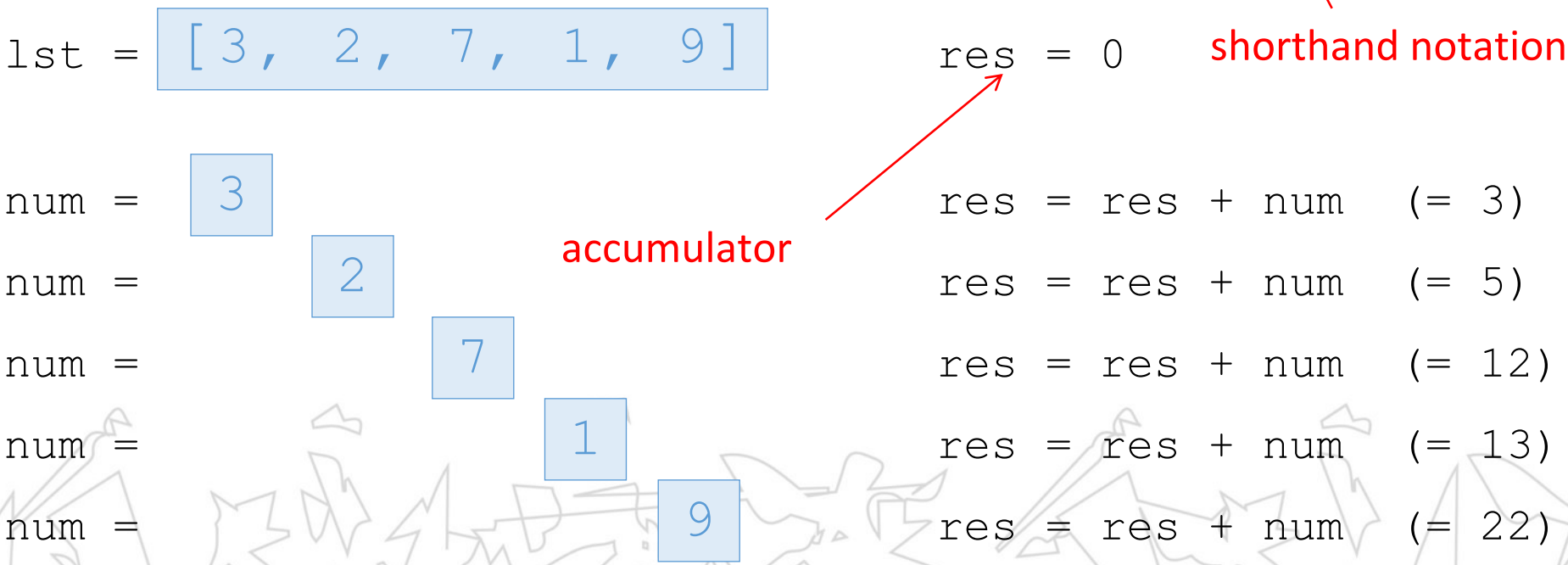
Accumulator loop pattern

Accumulating something in every loop iteration

For example: the sum of numbers in a list

```
>>> lst = [3, 2, 7, 1, 9]
>>> res = 0
>>> for num in lst:
        res += num
>>> res
22
```

shorthand notation

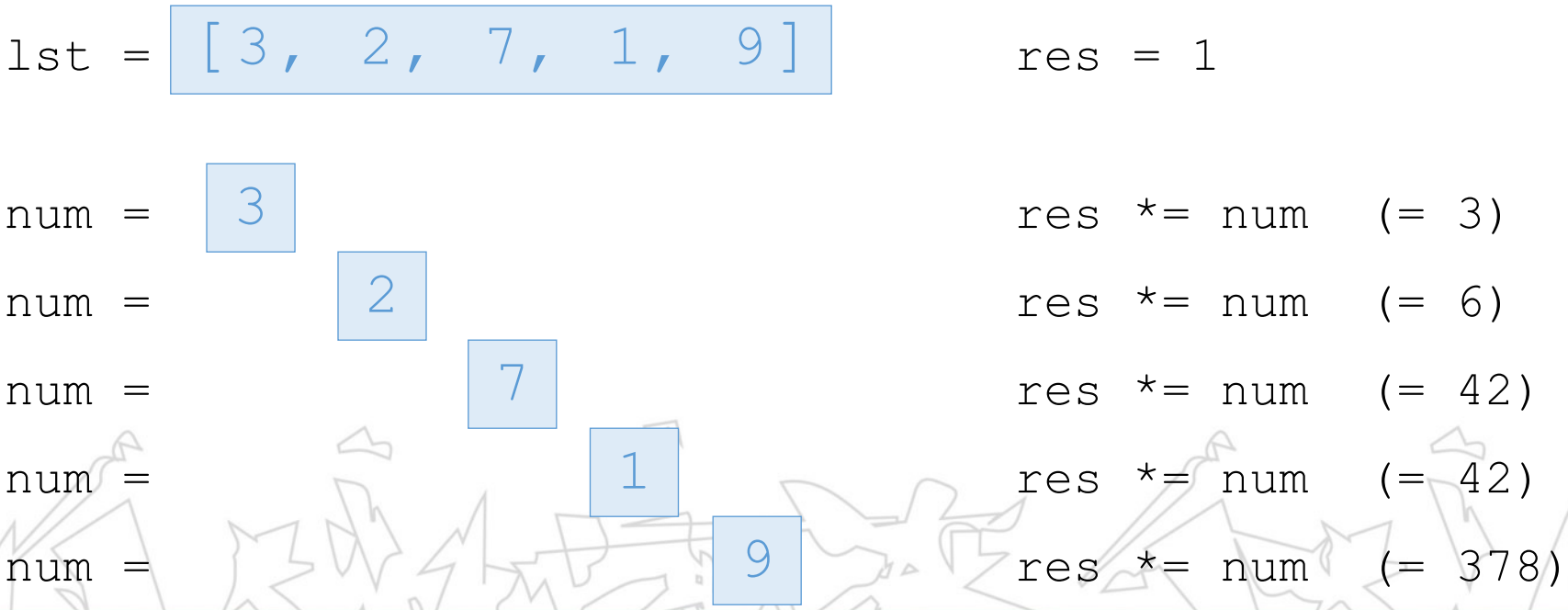


Accumulator loop pattern

Accumulating something in every loop iteration

What if we wanted to obtain the product instead?
What should `res` be initialized to?

```
>>> lst = [3, 2, 7, 1, 9]
>>> res = 1
>>> for num in lst:
        res *= num
```



Exercise

Write function factorial() that:

- takes a non-negative integer n as input
- returns n!

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 3 \times 2 \times 1 \quad \text{if} \quad n > 0$$
$$0! = 1$$

```
>>> factorial(0)
1
>>> factorial(1)
1
>>> factorial(3)
6
>>> factorial(6)
720
```

```
def factorial(n):
    'returns n! for input integer n'
    res = 1
    for i in range(2, n+1):
        res *= i
    return res
```

Infinite loop pattern

An infinite loop provides a continuous service

```
>>> hello2()
What is your name? Sam
Hello Sam
What is your name? Tim
Hello Tim
What is your name? Alex
Hello Alex
What is your name?
```

A greeting service

The server could instead be a time server, or a web server, or a mail server, or…

```
def hello2():
    '''a greeting service; it repeatedly requests the name
       of the user and then greets the user'''

    while True:
        name = input('What is your name? ')
        print(f'Hello {name}')
```

The `break` statement

The `break` statement:

- is used inside the body of a loop
- when executed, it interrupts the current iteration of the loop
- execution continues with the statement that follows the loop body.

```
for letter in "my_string":
    if letter == "i":
        break
    print (letter)
```

OutPut:

```
m
y
_
s
t
r
```

The `break` statement

Another example with `break` statement inside a function:

- is used inside the body of a loop
- when executed, it interrupts the current iteration of the loop
- execution continues with the statement that follows the loop body.

```
def cities2():
    lst = []

    while True:
        city = input('Enter city: ')

        if city == '':
            return lst

        lst.append(city)
```

```
def cities2():
    lst = []

    while True:
        city = input('Enter city: ')

        if city == '':
            break

        lst.append(city)

return lst
```

`continue` statement

The `continue` statement:

- is used inside the body of a loop
- when executed, it interrupts the current iteration of the loop
- execution continues with next iteration of the loop

```
# select only consonants
count = 0
for letter in "my_string":
   if letter in "aeiou":
         continue
   count +=1
print (count)
```

```
# select only count only vowels
count =0
for letter in "my_string":
   if letter not in "aeiou":
         continue
   count +=1
print (count)
```

Output:

```
8
```

Output:

```
1
```

Exceptions
Gracefully handling errors

- Unexpected errors might occur if we your code is working on the wrong type, undefined variable etc.
- Its possible to write programs that can handle selected errors or such errors.
- Python provides the try and except keywords to manage exceptions

```
while True:
        try:
                x = int(input("Please enter a number: "))
                break
        except ValueError:
                print("Oops! That was no valid number. Try again...")
```

Exceptions
Gracefully handling errors

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops! That was no valid number. Try again...")
```

- First, the ***try*** *clause* (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
- Other types of exceptions are available include RuntimeError, TypeError, and NameError

Exceptions
inspecting the exception

```
while True:
  try:
      x = int(input("Please enter a number: "))
      break
  except ValueError as  err:
       print(“err”)
       print("Oops! Try again...")
```