CS115

INTRODUCTION TO PROGRAMMING IN PYTHON

Strings, Selection Statements (Branching)

Strings

Strings are a sequence of case sensitive characters.

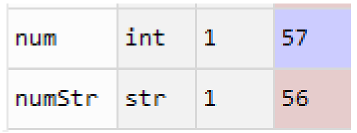
May be letters, special characters, spaces, digits.

Strings are enclosed in quotation marks or single quotes.

```
hi = "hello there"

ch = 'a'
```

Strings may contain numeric values, but the values are stored as Strings.

```
num = 57
numStr = "56"
```



num	int	1	57
numStr	str	1	56

Can compare strings with ==, >, < etc.

String Operations – len()

Python offers a number of functions that can be applied to a string.

We can find the length of strings, searching strings, finding substrings.

len(str): returns the number of characters in a string

```
    In [3]:len("abc")

    Out[3]: 3
```

String Operations – Indexing

Each character in a string has an index or position.

In Python, indexing starts from zero to indicate the first element of a string.

Square brackets are used to get the value at a certain index/position

```
s = "abc"
s[0] evaluates to 'a'
s[1] evaluates to 'b'
s[2] evaluates to 'c'
s[3] trying to index out of bounds, error
s[-1] evaluates to 'c'
s[-2] evaluates to 'b'
s[-3] evaluates to 'a'
```

```
index: 0 1 2 indexing always starts at 0

index: -3 -2 -1 last element always at index -1
```

String Operations – Indexing

Indexing can be used to extract individual characters from a string.

If the index given is outside the bounds of the string, an error message will be displayed.

```
s[3]
IndexError: string index out of range
```

If negative indexes are given, indexing starts from the end of the string.

```
s[-1]
Out[5]: 'c'

s[-2]
Out[6]: 'b'

s[-3]
Out[7]: 'a'

s[-4]
IndexError: string index out of range
```

String Operations – Slicing

Slicing is used to extract substrings of specified length.

If `s` is a string, the expression `s[start:end]` denotes the substring of `s` that starts at index `start` and ends at index (`end-1`).

Examples:

```
s = "hello world!"
s[0:len(s)]
Out[8]: 'hello world!'
s[0:len(s)-1]
Out[9]: 'hello world'
s[6]
Out[10]: 'w'
s[6:11]
Out[11]: 'world'

s[:]
Out[11]: 'hello world!'
```

String Operations – Slicing

Can slice strings using `[start:end:step]`

If you give two numbers, `[start:end]`, `step = 1` by default

You can also omit numbers and leave just colons

```
s = "abcdefgh"
s[3:6]          -> evaluates to "def", same as s[3:6:1]
s[3:6:2]        -> evaluates to "df"
s[::]           -> evaluates to "abcdefgh", same as s[0:len(s):1]
s[::-1]         -> evaluates to "hgfedcba", same as s[-1:-(len(s)+1):-1]
s[4:1:-2]       -> evaluates to "ec"
```

String Concatenation

String concatenation operator (+) : joins two strings.

```
name = "ana"
name = 'ana'
greet = 'hi' + name
greeting = 'hi' + " " + name
```

To join data of other types to string, you must first convert the data to a string using the str() function.

```
 "a" + 5
 TypeError: must be str, not int

"a" + str(5) -> "a5"
```

String Repetition Operator

Repetition operator (*): the expression `n * s`, where `n` is an integer value and `s` is a string, evaluates to a String with `n` repeats of `s`.

Just as 3 * 2 is equal to 2 + 2 + 2, the expression 3 * `"a"` is equivalent to

```
"a" + "a" + "a" ("aaa").
```

Example of operations performed on a string as defined in Python docs.

```
silly = 'hi' + " " + name * 3
```

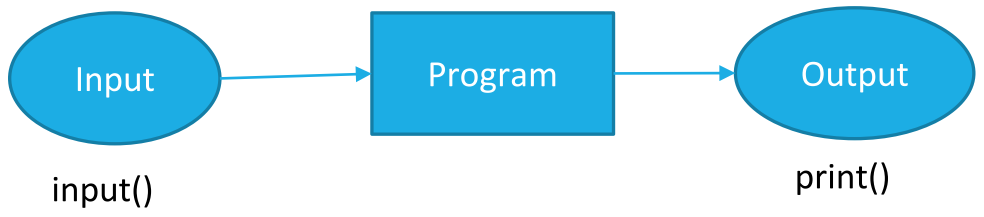
Error:

```
'a' * 'a'
```

```
TypeError: can't multiply sequence by non-int of type 'str'
```

Input

Python 3 has a function that can be used to get input directly from the user.

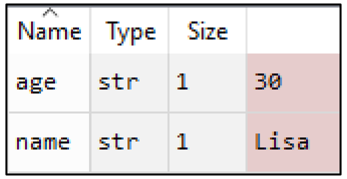


Example:

```
text = input("Type anything:") #Prints whatever is in quotes.
Type anything: hi              #User types something, hits enter.
print(3 * text)                #Binds that value to a variable.
hihihi
```

Input Strings and Type Casting

The `input()` function returns a string



Name	Type	Size	
age	str	1	30
name	str	1	Lisa

Example:

```
name = input("Enter your name: ")
Enter your name: Lisa
print(type(name))
<class 'str'>
age = input("Enter your age: ")
Enter your age: 30
print(type(age))
<class 'str'>
```

If working with numbers, you must cast the return string

```
num = int(input("Type a number... "))
 print(5*num)
```

Input/Output Example:

Note the difference between outputting string using the concatenation operator versus the comma in the print statements below.

```
name = input("Enter your name: ")
Enter your name: Lisa


print("Are you really " + name +"?")
Are you really Lisa?


print("Are you really",name,"?")
Are you really Lisa ?
```

Character Encoding

For a computer, there is no text, only bytes.

Text is an interpretation, a way to visualize bytes in human-readable form. And if you interpret bytes wrong, you get strange looking characters, or an error.

Character encoding is one specific way of interpreting bytes: It's a look-up table that says, for example, that a byte with the value 97 stands for 'a'.

By default, Python assumes the character encoding format, UTF-8.

To define a source code encoding, a magic comment must be placed into the source files either as first or second line in the file, such as:

`# -*- coding: utf-8 -*-` ***or*** `coding = < ISO-8859-9 >`

Python has built-in functions: **`ord()`** and **`chr()`**

`ord(char)` `to get numerical code for character char` **`chr(num)`** `to get corresponding character for integer num`

Example:

Write a script that inputs the hourly wage of an employee and the number of hours worked and calculates and displays the employee’s salary.

```
hourly_wage=float(input('Enter your hourly wage: '))

hours=int(input('How many hours have you worked? '))

salary=hourly_wage * hours

print('Your salary is: ' + str(salary) + ' $')
```

Sample Run:

```
Enter your hourly wage: 12.5

How many hours have you worked? 37

Your salary is: 462.5 $
```

Flow of Control

The order of statement execution is called the *flow of control*

Unless specified otherwise, the order of statement execution through a method is linear: one after another

Some programming statements allow us to make decisions and perform repetitions

These decisions are based on *boolean expressions* (also called *conditions*) that evaluate to true or false

Conditional Statements and Boolean Expressions

A *conditional statement* lets us choose which statement will be executed next

Conditional statements are implemented using if/else statements.

Boolean expression is any expression that evaluates to true of false.

Boolean expression may include arithmetic expressions, logical and relational operators, etc.

Comparing Values – Relational Operators

A condition often uses one of Python's *equality operators* or *relational operators*, which all return boolean results:

`==`	equal to
`!=`	not equal to
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

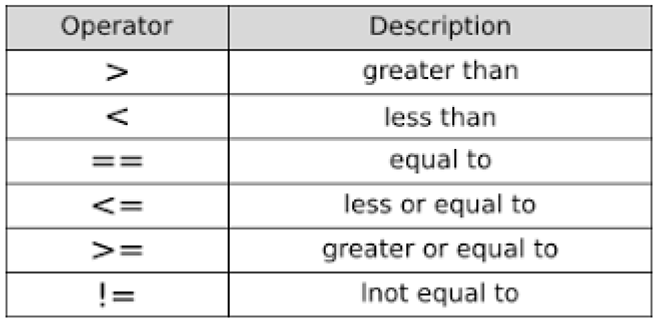
Note the difference between the equality operator (==) and the assignment operator (=)

COMPARISON OPERATORS ON int, float, string

Assume i and j are variable names

The comparisons below evaluate to a Boolean

```
i > j
i >= j
i < j
i <= j
i == j =>  equality test, True if i is the same as j
i != j =>  inequality test, True if i not the same as j
```



Operator	Description
>	greater than
<	less than
==	equal to
<=	less or equal to
>=	greater or equal to
!=	Inot equal to

COMPARISON OPERATORS – in, not in

Python has two operators that can also be applied to strings, in, not in

These operators are used to test for membership in a group.

Usage:

<string> <operator> <string>

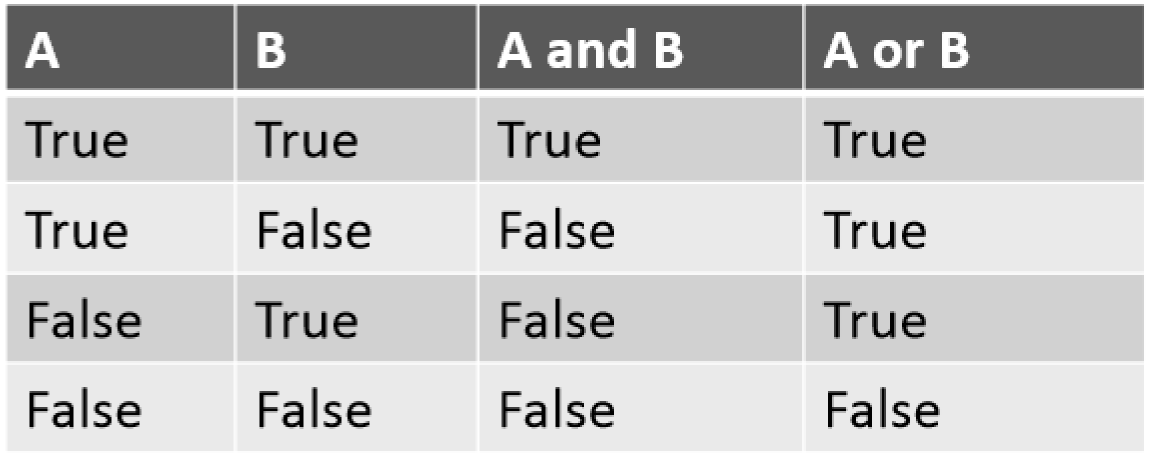
Example

```
'way' in 'John Wayne'
Out[1]: False

'way'.lower() in 'John Wayne'.lower()
Out[2]: True

'was' not in 'I was happy'
Out[3]: False

'was' not in 'I am happy'
Out[4]: True
```

LOGIC OPERATORS ON bools

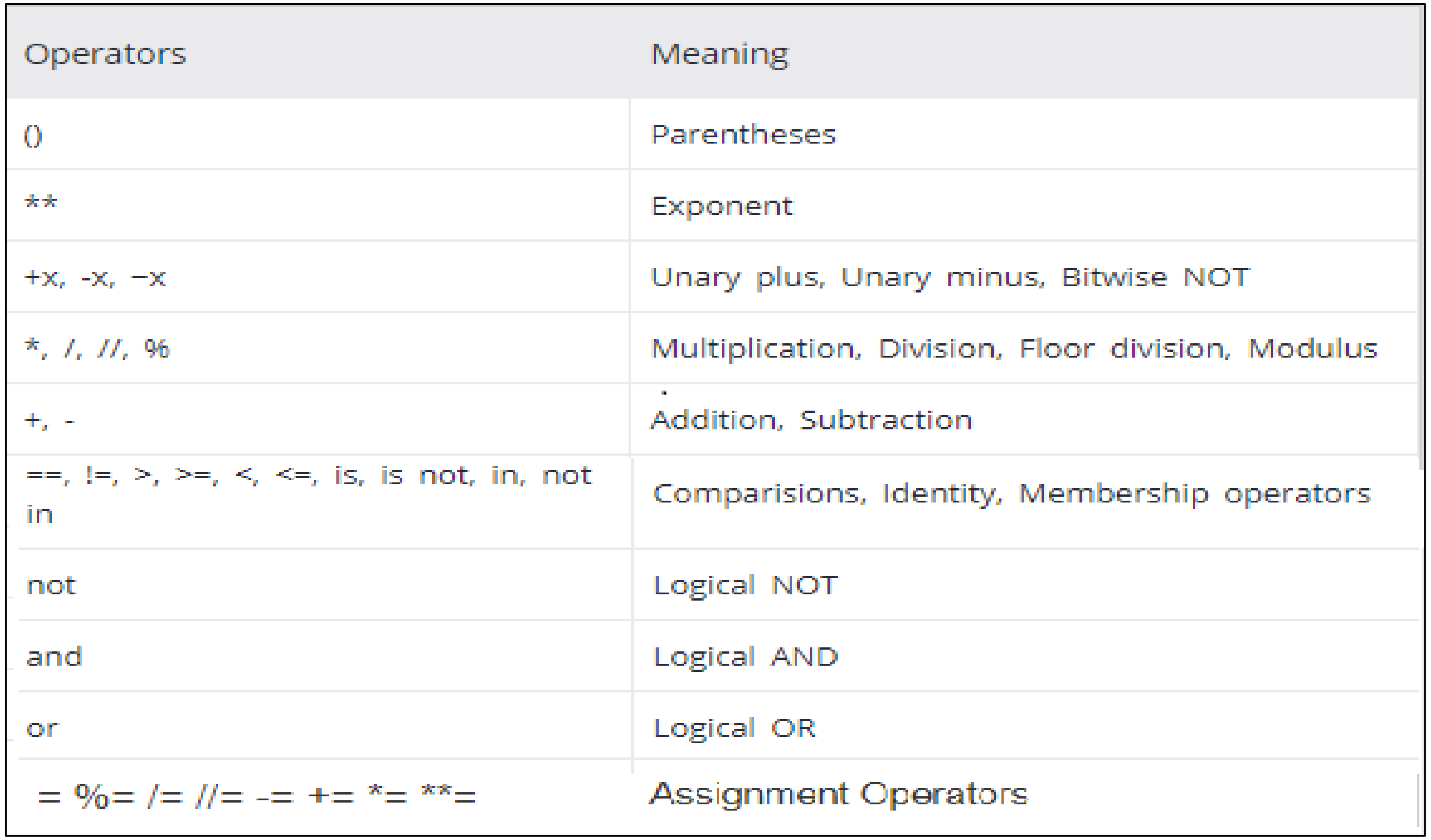
Assume a and b are variable names (with Boolean values)

not a	->	True if a is False False if a is True
a and b	->	True if both are True False if either is False
a or b	->	True if either or both are True False if neither are True



A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

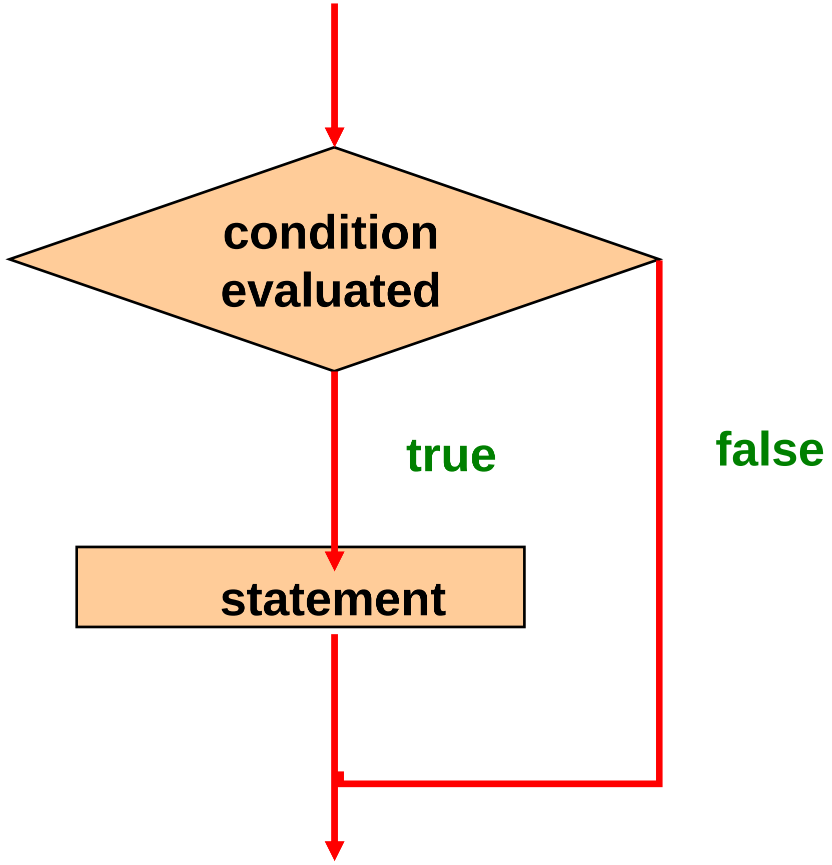
Python Operator Precedence



Operators	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisions, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR
= %= /= //= -= += *= **=	Assignment Operators

Branching - IF Statements

```
if <condition>:
        < statement >
        < statement >
        ...
```



Branching - IF Statements

`if` is a Python reserved word

The *condition* must be a boolean expression. It must evaluate to either true or false.

```
if  condition :
    statement
```

If the *condition* is true, the *statement* is executed.
If it is false, the *statement* is skipped.

Indentation

Semantically meaningful in Python.

Most other programming languages ignore whitespace, and indentation is for the reader of the code.

Python is unusual in that indentation is required and is strictly enforced during execution.

Example:

```
x = 5
if x > 10:
   print( x )
print("done")
```

Output:

```
    done
```

```
x = 5
if x > 10:
   print( x )
   print("done")
```

Output: -

```
x = 5
if x > 10:
print( x )
```

Output: ERROR

Branching - IF Statements (Example)

```
if(num % 2) == 0 :
  print ( ‘num is even’)

#non-zero values evaluate to True
if 5:
  print( "Condition is true" )

#zero values evaluate to false
if 0:
  print( "Condition is false" )
```

Example: Write a script that inputs the hourly wage of an employee and the number of hours worked and calculates and displays the employee’s salary. The employee will receive 100$ if he/she has worked more than 40 hours.

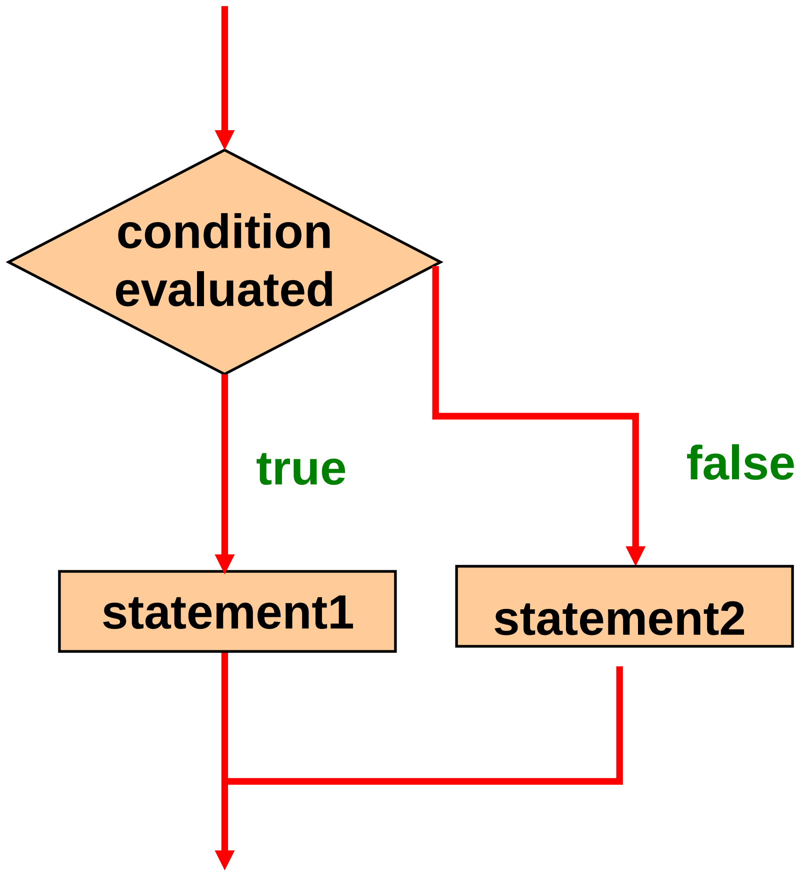
```
hourly_wage=float(input('Enter your hourly wage: '))
hours=int(input('How many hours have you worked? '))
salary=hourly_wage * hours
if hours > 40:
    salary +=100
print('Your salary is: ' + str(salary) + ' $')
```

Sample Run:

```
Enter your hourly wage: 12.5

How many hours have you worked? 50

Your salary is: 725.0 $
```

if/else Statement

```
if <condition>:
       < statement >
       < statement >
       ...
else:
       < statement >
       < statement >
       ...
```



if/else Statement

An *else clause* can be added to an `if` statement to make an *if-else statement*

```
if condition:
    statement1
else:
    statement2
```

If the *condition* is true, *statement1* is executed; if the condition is false, *statement2* is executed

One or the other will be executed, but not both

Example:

```
x , y = 2 , 3
if x > y:
      print('x is larger')
else:
       print(‘x is not larger')
print('Done')
```

Output:

```
x is not larger
Done
```

Example: Write a script that inputs the hourly wage of an employee and the number of hours worked and calculates and displays the employee’s salary. The employee will receive 50% more for each extra hour he/she has worked above 40 hours.

```
hourly_wage=float(input('Enter your hourly wage: '));
hours=int(input('How many hours have you worked? '));
if hours < 40:
    salary=hourly_wage * hours
else:
     salary=40*hourly_wage + (hours-40)*hourly_wage*1.5
print('Your salary is: ' + str(salary) + ' $')
```

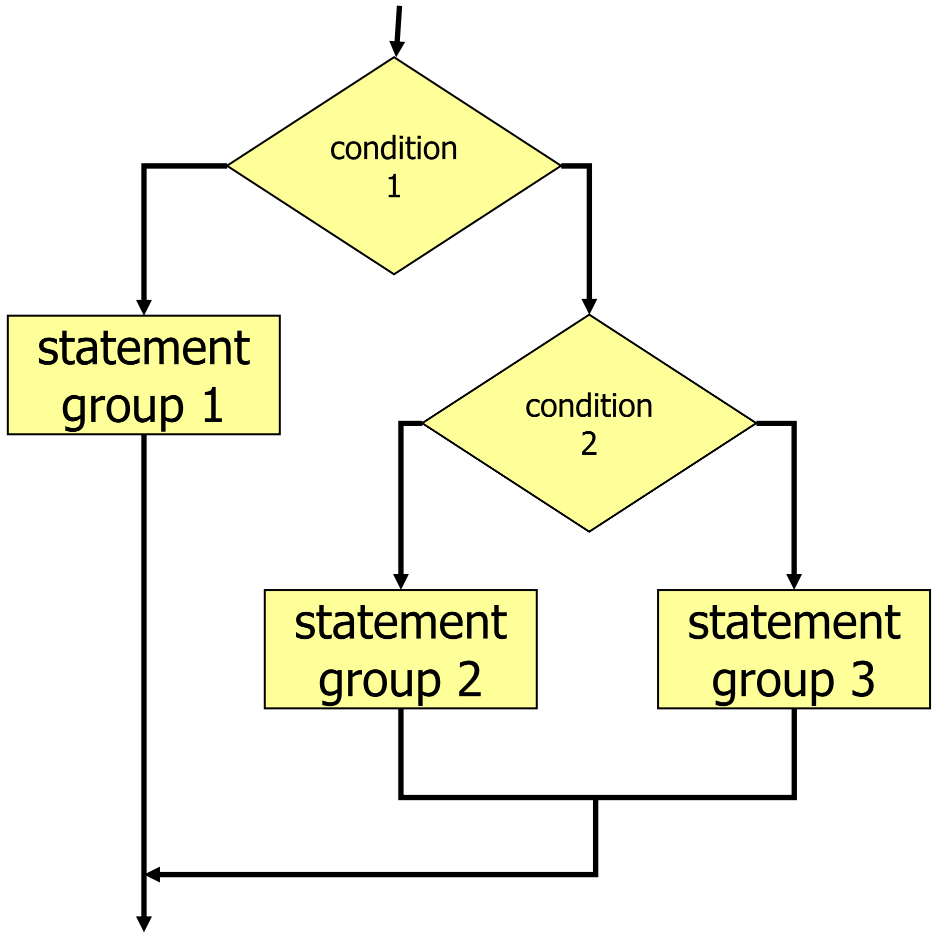
Sample Run:

```
Enter your hourly wage: 20

How many hours have you worked? 60

Your salary is: 1400.0 $
```

See: `02_employee.py`

Nested IF Statements (if/elif)

```
if <condition>:
    < statement >
    < statement >
    ...
elif <condition>:
    < statement >
    < statement >
    ...
else:
    < statement >
    < statement >
    ...
```



Example:

```
x , y = 2 , 2
if x > y:
     print('x is larger')
elif x < y:
        print('y is larger')
else:
        print('x is equal to y')
print('Done')
```

Output:

```
x is equal to y

Done
```

Assigning letter grades to students

Range	Grade
100 >= grade >90	A
90 >= grade > 80	B
80 >= grade > 70	C
70 >= grade > 60	D
grade <= 60	F

See: `02_letter_grade.py`

Exercises (02_exercises.py)

1. Input a number from the user, and output '[number] is divisible by 5 or 7' if the number is divisible by 5 or 7.
2. Write a Python program that initializes a number between 1-9 and prompts the user to guess the number. If the user guesses incorrectly display the message, "Wrong Guess!", on successful guess, user will see a "Well guessed!" message.
3. Input a word and give an appropriate message if the word has the same letter at the beginning and the end of the word or not.
4. Input a user's height (in metres) and weight (in kgs) and calculate his/her BMI and output the BMI category according the following:

```
    bmi >= 30           Obese
    25 <= bmi <30       Overweight
    20 <= bmi <25       Normal
    bmi < 20            Underweight
```

Terms of Use