2. Conditionals and loops

1. Boolean expressions

Boolean expressions

- A *boolean expression* is an expression that is either true or false.
- For example, the following examples use the operator `==`, which compares two operands and produces `True` if they are equal and `False` otherwise:

```
>>> 5 == 5
True
>>> 5 ==6
False
```

- `True` and `False` are special values that belong to the type `bool`; they are not strings:

```
>>> type(True)
<class 'bool'>

>>> type(False)
<class 'bool'>

>>> type(7==7)
<class 'bool'>
```

Relational operators

- Python has a set of "**Operators**" that can be used to ask mathematical questions.

Symbol	meaning
> and <	larger and smaller
>=	greater than or equal to
<=	less than or equal to
==	equals
!=	not equal to

2. Logical operators

- There are three logical operators: `and`, `or`, and `not`. Their semantics is similar to their meaning in English.

`and` operator

- `<boolean expression A> and <boolean expression B>`. It is `True` if *both* boolean expressions are `True`.
- For example, `x > 0` and `x<10` is true only if `x` is greater than 0 and less than 10. We try it in terminal.

```
>>> x = 1
>>> x > 0 and x < 10
True
```

- When `x = 1`, both `x > 0` and `x < 10` are `True`. Hence the final result is `True`.
- If we change `x` to -1, we get `False` since now `x > 0` is `False`. See the code below.

```
>>> x = -1
>>> x > 0 and x < 10
False
```

`or` operator

- `<boolean expression A> or <boolean expression B>`. It is `True` if *either or both* of the boolean expressions is `True`. For example,

```
>>> x = 1
>>> x > 0 or x < 10
True
```

- In the above code, both `x > 0` and `x < 10` are `True`. Hence the final result is `True`.
- If we change `x` to -1, we still get `True`. Now `x > 0` is `False` but `x < 10` is `True`. Hence the final result is `True`. See the code below.

```
>>> x = -1
>>> x > 0 or x < 10
True
```

- Let us try the following. The result is `False` now since both boolean expressions are `False`.

```
>>> x = 2
>>> x > 10 or x < 0
False
```

`not` operator

- `not` operator negates a boolean expression. `not <boolean expression>` is `True` only is the boolean expression is `False`.
- For example,

```
>>> x = 1
>>> not (x > 1)
True

>>> x = 2
>>> not (x > 1)
False
```

number as boolean value

- 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`

```
>>> 2 and True
True

>>> 0 and True
False
```

3. Conditional statement: `if` statement

- When we are trying to finish some tasks, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability.

- For example, given a number, we would like to state the number is positive if it is positive. We can use a `if` statement.

```
x = float(input("Please input a number: "))

if x>0:
    print("It is a positive number.")
```

- The `if` statement compares x with 0. If the condition of `x > 0` is met (the boolean expression is `True`), the `print` statement is executed; otherwise, not.
- `if` statements have the same structure as function definitions: a header followed by an *indented body*.

Number of statement in the indented bodies

- There is no limit on the number of statements that can appear in the body, but there has to be at least one.
- Occasionally, it is useful to have a body with *no* statements (usually it is used as a place keeper for code you haven't written yet and will be finished later). In that case, you can use the *pass statement*, which does nothing.

```
x = 1
if x>0:
    pass
```

4. Alternative execution: Control Flow, `else`, and `elif`

- the code

```
x = float(input("Please input a number: "))

if x>0:
    print("It is a positive number.")
```

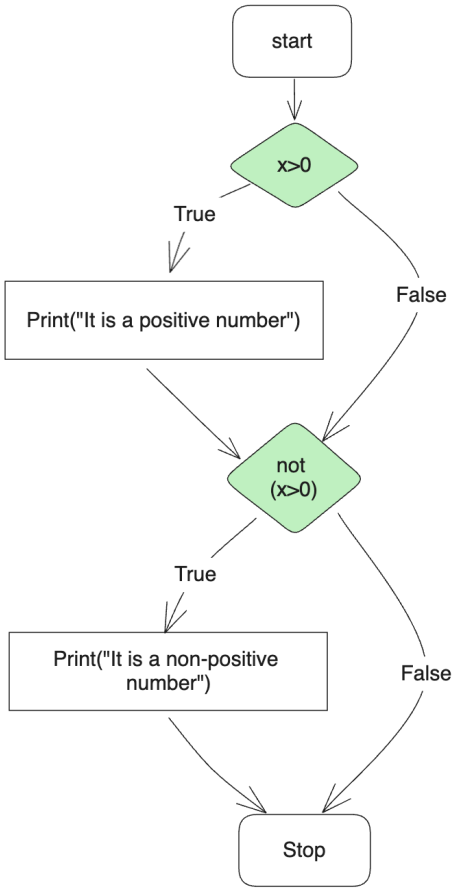
- The above code can only state "It is a positive number." if we input a positive number. If we input zero or a negative number, it shows nothing.

- Now let us enrich the code, which can judge whether the number inputted is positive or non-positive. We can simply add one more `if` statement.

```
x = float(input("Please input a number: "))

if x>0:
    print("It is a positive number.")
```

```
if not (x>0):
    print("It is a non-positive number.")
```

- Here is how we are providing two `if` statements. First, the first `if` statement is evaluated. Then, the second one is evaluated. The flow of decisions is called "control flow", which can be illustrated as follows.

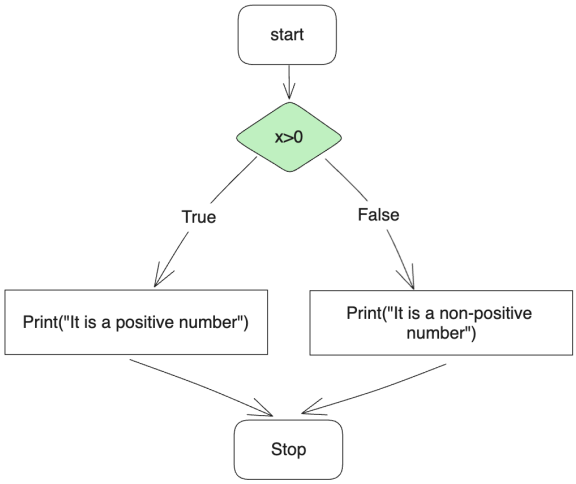


`else` statement

- This program can be improved since if the first `if` statement is `False`, x must be not greater than 0. Hence we don't need to do the second `if` statement. Instead, we use `else`.

```
x = float(input("Please input a number: "))

if x>0:
    print("It is a positive number.")
else:
    print("It is a non-positive number.")
```

- The new control flow can be illustrated as follows.

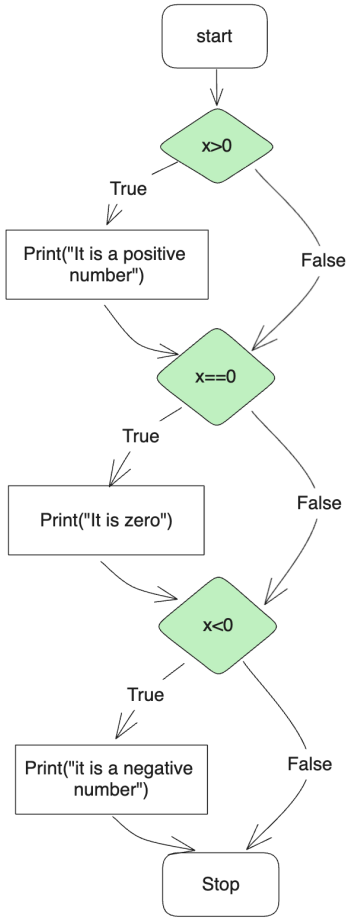


- `if-else` statement is useful for the two options that are mutually exclusive.

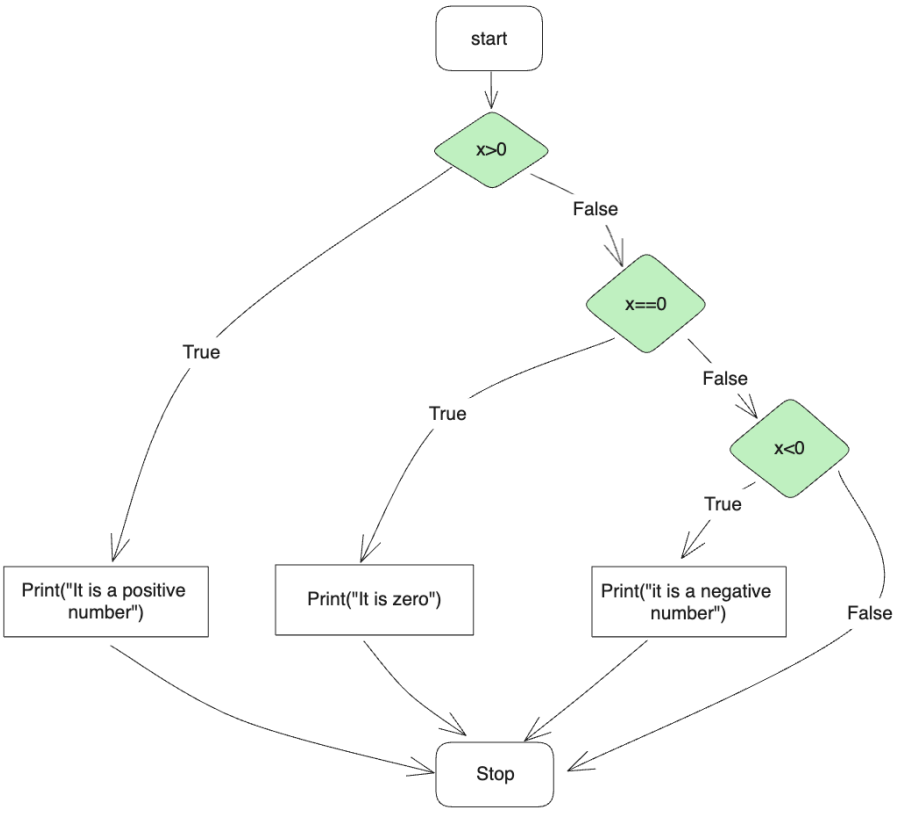
`elif` statement

- Sometimes we may have more than two options. For example, we would like to determine whether a number is (1) positive, (2) zero, and (3) negative.
- For sure, we can use three `if` statements like this.

```
x = float(input("Please input a number: "))

if x>0:
    print("It is a positive number.")

if x==0:
    print("It is zero.")

if x<0:
    print("It is a negative number.")
```



- The control flow can be illustrated as follows.
- In the diagram, we can see we always have to answer the next question whenever the answer of the questions is `True` or `False`. Apparently, If `x>0` is `True`, it must be `False` for the next two questions. Hence it should be improved. The desired control flow should be as follows.



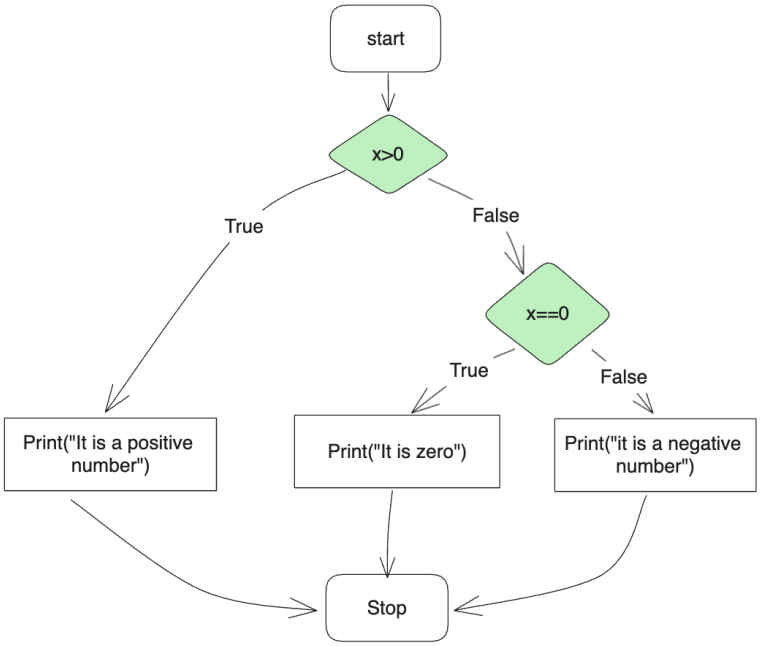
- We can use `if-elif` statements to implement this control flow.

```
x = float(input("Please input a number: "))

if x>0:
    print("It is a positive number.")
elif x==0:
    print("It is zero.")
elif x<0:
    print("It is a negative number.")
```

- Further improvement. Since the last question is mutually exclusive to the first two questions. So we don't necessarily need to ask the last question, but just use `else` statement, which makes the program more efficient.

```
x = float(input("Please input a number: "))

if x>0:
    print("It is a positive number.")
elif x==0:
    print("It is zero.")
else:
    print("It is a negative number.")
```

- The code can be represented as the following diagram.



5. `and`, `or` in `if` statement

- `and` and `or` can be used to connect to conditions together.
- `and` example

```
x = int(input("Please input an integer: "))

if x > 0 and x%2==0:
    print("x is a positive even number.")
else:
    print("x is not a positive even number.")
```

- In the above code, when the two conditions are satisfied, the code executes the `if` statement; otherwise, `else` statement.
- Similar to `and`, `or` can be used in the expression where `if` statement is executed if one of the two conditions is satisfied. For example,

```
x = int(input("Please input an integer: "))

if x > 0 or x <0:
    print("x is not zero.")
else:
    print("x is zero")
```

  - We could improve the code as follows:

```
x = int(input("Please input an integer: "))

if x!=0:
    print("x is not zero.")
else:
    print("x is zero")
```

6. Pythonic `if`

- 'Pythonic' code means the code only seen in Python programming.

```
x = int(input("Please input an integer: "))

b = True if x!=0 else False

if b:
    print("x is not zero.")
else:
    print("x is zero")
```

- Notice that this return statement in our code is almost like a sentence in English. This is a unique way of coding only seen in Python.

7. Repetitions and Loops

- Sometimes we need to repeat something more than one time. Let us begin with the following code:

```
print("Hello")
print("Hello")
print("Hello")
```

- Running the code above, we will notice that the program says "Hello" three times.
- As a programmer, when you would like to do something over and over again, you may need to consider how we can improve your code. Three times are fine. How about 1000 times? Do you want to type `print(Hello)` 1000 times?
- For sure, not! Python inventor (or any other programming language inventor) created loops, which enable us to create a block of code to execute repetitions.
- *Loops* are a way to repeat something over and over again. It is also called *iteration*.

8. `while` loops

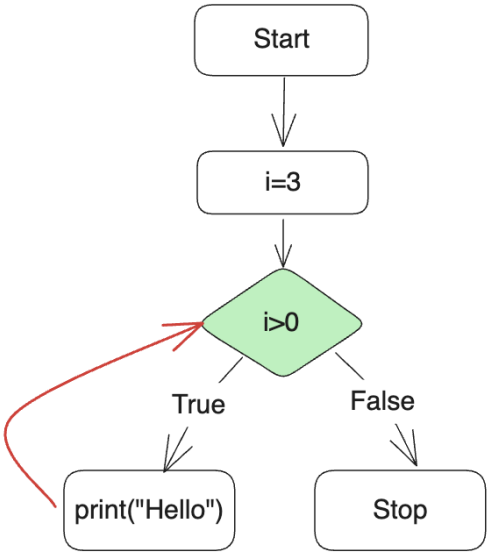
- The `while` loop is nearly universal throughout all programming languages. You can see it in C, C++, Java, et. al.
- The syntax is just like `if`:

```
while <condition>:
    <block>
```

- Let us use `while` loops to print `Hello` three times.

```
i = 3
while i>0:
    print("Hello")
```

- Run this program. This program really prints "Hello". However, it prints too many "Hello" and will *never stop!*
- You can press `control+c` keys to stop the program, when you get stuck in Python.
- Let us use the following diagram to show how `while` loops work.

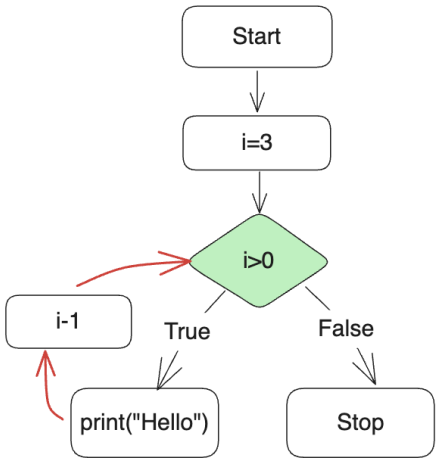


  - The program examines whether the condition `i>0` is satisfied. If true, it will print "Hello"; stop otherwise. In our program, `i` value is 3, indicating `i>0` is True. So the program will print "Hello".
  - After that, it will go back to the condition again to see whether the condition is still satisfied. If True, continue. Then repeat this process over and over again.
  - You can see the condition `i>0` is always satisfied since `i` value is 3.

- Try to think how to solve this problem.
- We can change the `i` value, i.e., update `i` value, by code `i=i-1`. Try the following program.

```
i = 3
while i>0:
    print("Hello")
    i = i - 1
```

- It works now. It prints "Hello" three times. The above `while` loops can be expressed using this diagram.



- Sure, we can change `i=i-1` to `i-=1` (decrement), which makes the program more concise.

```
i = 3
while i>0:
    print("Hello")
    i-=1
```

- Above, we are starting `i` at three, then two, one. It is not so natural for most humans. We like count forward, 1, 2, 3, ... It's best practice in programming to begin counting with zero. Hence, we can improve the code as follows.

```
i = 0
while i<3:
    print("Hello")
    i+=1
```

9. List basics

List

- List is a container in Python. A list is a sequence of values, which can be any type.
- We can define the following lists.

```
names = ["Jin", "Mark", "Neil"]
numbers = [3, 8, 8]

print(names)
print(numbers)
```

Getting access to the elements

- We can get access to the elements in a list by using the indices. Note that the index starts with 0.

```
names = ["Jin", "Mark", "Neil"]
numbers = [3, 8, 8]

print(names[0])
print(names[1])
print(names[2])

print(numbers[0])
print(numbers[1])
print(numbers[2])
```

Updating the elements

- We can update the elements by assigning new values.

```
numbers = [3, 8, 8]
numbers[2] = 5

print(numbers)
```

`range()` function

- The function `range()` can give us a range of number.

```
numbers_range = range(3)
numbers_list = list(numbers_range)

print(numbers_list)
```

- The output is `[0, 1, 2]`

len() function to obtain length

- The function `len()` returns the length of a list.

```
numbers = [3, 8, 8]
l = len(numbers)
print(l)
```

- The output is 3.

10. `for` loops

- `for` loops are another statements which can be used for do something repeatedly. The following program is to print "Hello" three times using `for` loops.

```
for i in [0, 1, 2]:
    print("Hello")
```

- The output is just the same as `while` loop statements above.
- In the above program, we traverse a list `[0, 1, 2]`. First, `i` takes the first value in the list, which is `1`; then execute the indented body, that is `print("Hello")`. After that, `i` takes the second value in the list, which is `2`; then execute the indented body again. After that, continue traversing the list and execute the indented body.
- We can improve the code by using `range()`:

```

```

```
for i in range(3):
    print("Hello")
```

- Our code can be further improved. Notice, we never use `i` explicitly in the indented body. The usage of `i` value is just for the iterations of the loop. In Python, if such a variable does not have any other significance in our code, we can simply represent this variable as a single underscore `_`. Therefore, you can modify your code as follows:

```
for _ in range(3):
    print("Hello")
```

- Now let us use `for` loops to print elements in a list. Since `for` loop traverses the list, we can use the following code.

```
for i in ["Jin", "Mark", "Neil"]:
    print(i)
```

- First, `i` takes the first value in the list, which is “Jin”; then execute the indented body, that is `print(i)`, where `i` value is "Jin". Then, `i` takes the second value, so on so forth.
- We can make the code more elegant.

```
names = ["Jin", "Mark", "Neil"]
for i in names:
    print(i)
```

Traversing a list via indices in `for` loop

- We can also traverse a list via indices in `for` loop. For example,

```
names = ["Jin", "Mark", "Neil"]
n = len(names)
for i in range(n):
    print(names[i])
```

- It seems redundant since we can traverse a list directly. However, it is very useful when we want to update or write the elements.

```
numbers = [3, 8, 8]
n = len(numbers)

for i in range(n):
```

```
    numbers[i] = numbers[i] * 2

print(numbers)
```

11. `continue` and `break`

break

- In a previous question, we want to give letter grade given a score (between 0 and 100). However, we didn't give a program to let user correct the input. Let's create it by using loops.

```
while True:
    n = int(input("Please input a score (0-100): "))
    if 0<=n<=100:
        break

print(n)
```

- In the above program, `break` *tells Python to "break out" of a loop early, before it has finished all of its iterations*.
- This program will reprompt the use with "Please input a score (0-100): " when the user's input is not within the range (0-100).
- If the input is in the range, the program will execute `break` to exit the loops.

continue

- Different from `break`, *`continue` tells Python to go to the next iteration for a loop and ignore the code after `continue`*. For example,

```
for i in range(4):
    if i == 1:
        continue

    print(i)
```

- Output is

```
0
2
3
```

- There is not 1 in the output since when `i` value is 1 Python executed `continue` which results in ignoring the code after Line 3.

Practice Questions - 1: conditionals

1. Create a function named `is_even`, which returns `True` if inputting an even number; return `False`.

2. In a right triangle, the lengths of the sides are a, b and the hypotenuse is c. Pythagoras theorem says that $a^2 + b^2 = c^2$. Write a function named *check_pythagoras* that takes parameters, a, b and c, and checks to see if Pythagoras theorem holds. If it holds, the program should print "Pythagoras theorem is satisfied.". Otherwise, the program print, "No, Pythagoras theorem isn't satisfied."

3. Use `if-elif-else` statements to finish this question. Professors give letter grade based on the score a student gets in an exam. Write `Python` code which can print a letter grade given a score value.

Score	Letter grade
90-100	A
80-89	B
70-79	C
60-69	D
<60	F

Practice Questions - 2: functions

1. Create a function named `get_maximum` taking a list of numbers, which can return the maximum number in the list.

2. Create a function taking a list, which can print the even indices values, i.e., the 2nd value, 4th value, 6th value, ... For example, given a list `x = ['a', 'b', 'c', 'd']`, the output should print 'b' and 'd'. (try to use `continue` in the loop)

3. Calculate the sum of 1 to n.

    - Create a function named `get_number` which returns a positive number. In the function you use `input` function to let user input a positive integer. If user inputs a negative number, reprompt the user with "Please input a positive number: " .
    - Create a function named `get_sum` taking a number n and returning the sum of 1 to n.
    - Create a `main` function which calls the two functions above to calculate the sum of 1 to n.

4. Given a list of numbers, try to print the numbers starting with the first one and stop when the sum of the numbers collected is larger than 100.

Summing up

- Conditionals
- Logical operators: `or`, `and`, `not`
- `if` statements
- Control flow, `elif`, `else`
- Pythonic coding
- `while` and `for` loops

- `continue` and `break` in loops