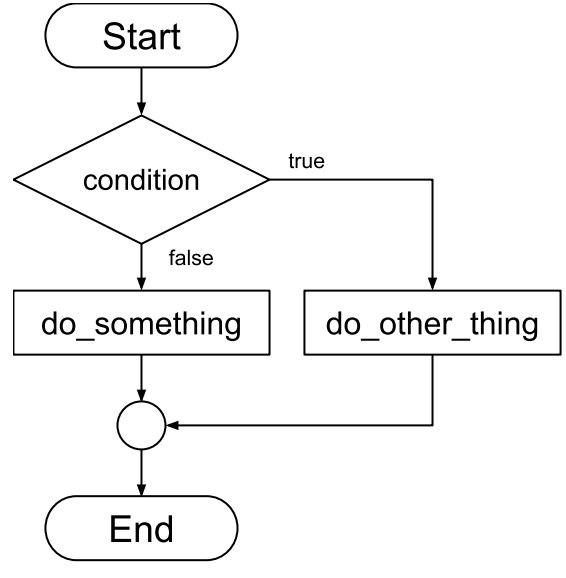
Python Workshop

2. Control Structures

Quick jump: Branching | while-loop | For-loop | Advanced topics

Concepts: Branching

- A program could deviate from the main flow under a certain condition.
- We need to define a condition that evaluates to either true or false to control it.



Branching example

- Example:

```
val = int(input())
if val > 0:
    print(val, 'is positive!')
```

- This reads an integer, then print a message if `val` is greater than 0.
- Further discussion follows.

Conditions

- Remember that a variable can store a boolean value, `True` or `False`.

```
y = True
n = False
```

- A condition is an expression that evaluates to either `True` of `False`. For example, `val > 0` is a condition.

Program block (1)

- Take a look at the example again:

```
val = int(input())
if val > 0:
    print(val, 'is positive!')
```

- Notice that the last statement is **indented one level**.
- Indentation defines a programming block in Python.
- Such block must follow a line ending with a colon `:`.

Program block (2)

- Now, let's see another example:

```
val = int(input())
if val > 0:
    if val > 10:
        if val > 100:
            print(val, 'is huge!')
        print(val, 'is big!')
    print(val, 'is positive!')
```

- Remember that indentation defines a block in python.
- What happens if the input value is `5`, `50`, or `500`?

Self-learning topics (~60min)

- Branching (**if**-statement)
- Looping until condition fails (**while**-loop)
- Looping through a list of values (**for**-loop)
- Advanced topics
  - Advanced loop control
- You are encouraged to copy-and-paste the code to Spyder and test them.

Comparison operators

Comparison operators

- **Comparison operators** `<`, `<=`, `==`, `>=`, `>`, and `!=` (not equal) can be used to produce a boolean value. For example:

```
a = 10
b = 3
print(a, '<', b, 'is', a < b)     # output 10 < 3 is False
print(a, '!=', b, 'is', a != b)   # output 10 != 3 is True
```

Locigal operators

- We can use `and`, `or`, and `not` as the logical operators on boolean values:

```
a = 10
b = 3
c = 5
print(a > b and c > b)   # output True
print(a < b or a < c)    # output False
print(not a < b)         # output True
```

if-statement

if-statement

- An **if-statement** will only be executed when the condition evaluates to `True`. For example:

```
number = int(input('Input a number: '))
if number < 0:
    number = -number
print('Absolute value of your number is', number)
```

Output:

```
Input a number: 1
Absolute value of your number is 1
```

```
Input a number: -1
Absolute value of your number is 1
```

Revision: program blocks

- Remember that **indentation** is used to define a block. Consider the following two sets of code:

```
a = 0
if a > 1:
    a = a + 10
    a = a + 10
print("a is", a)
```

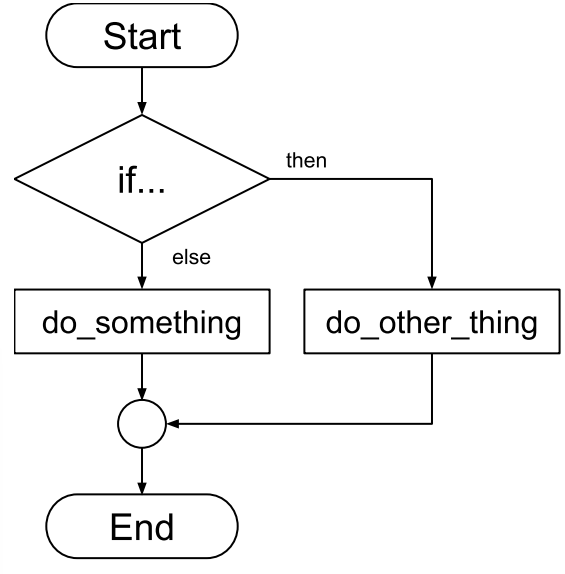
```
a = 0
if a > 1:
    a = a + 10
a = a + 10
print('a is', a)
```

- What are the outputs of the two codes above?

if-else

- We can use `else` after `if` to specify alternative action when the condition does not match.

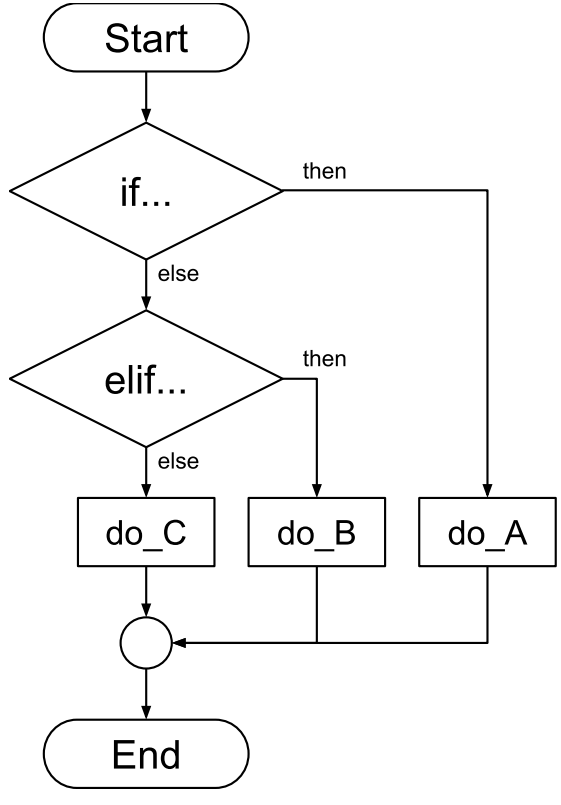
```
number = int(input('Input a number: '))
if number < 0:
    print('Your number is negative.')
else:
    print('Your number is not negative.')
```



elif

- We can use `elif` (else if) to specify another conditions and the corresponding actions.

```
number = int(input('Input a number: '))
if number < 0:
    print('Your number is negative.')
elif number == 0:
    print('Your number is zero.')
else:
    print('Your number is positive.')
```



- Note that a **condition** is needed for the use of `elif`.

Exerises (1)

Exercise 1 (1)

- Write a program that reads one integer that represents a year, and output `True` or `False` to indicate if it is a leap year or not.
  - If it is divisible by 4, it is a leap year,
  - except if it is divisible by 100, it is not a leap year,
  - except if it is divisible by 400, it is a leap year.

Exercise 1 (2)

- Sample input/output as follow:

Input	Output
1900	False
1924	True
1925	False
2000	True
2016	True
2018	False

Compound assignment operators

Compound assignment operators

- **Compound assignment operator** `+=` allows the addition operation to be applied to the variable itself.
- For example, `a += 10` will add 10 to `a`, the same as `a = a + 10`.

```
a = 100
a += 10
print('a =', a)
```

The output is `a = 110`.

Compound assignment operators (2)

- All arithmetic operators has their corresponding compound assignment operator. For example:

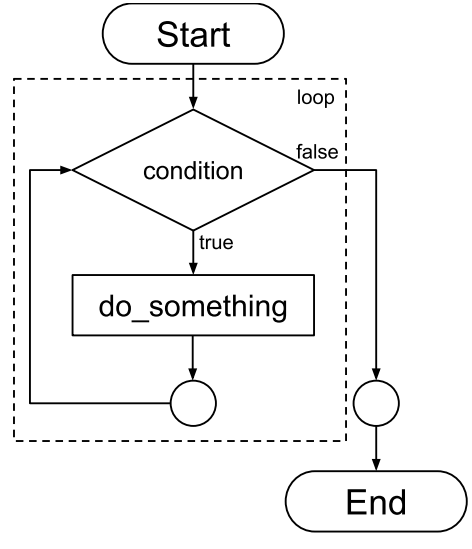
```
a = 100
a += 10
a -= 5
a *= 1.2
a /= 3
print(a)
```

The output is `42.0`.

while-loop

Concepts: Looping

- A program could repeatedly execute some code segment
- A condition is needed to control the number if the loop should be continued.



while-loop (1)

- **While-loop** can be used to execute codes repeatedly.

```
number = int(input('Input a number: '))
while number > 0:
    print(number)
    number -= 1
print('done.')
```

- Remember how a block is defined in Python? Can you point out which part above is inside the **while-loop**?

while-loop (2)

- The part `while number >0:` specifies a while-loop, which will be executed when the condition `number > 0` is `True`.

```
while number > 0:
```

- Only the two lines below `while` is inside the while-loop.

```
    print(number)
    number -= 1
```

while-loop (3)

```
number = int(input('Input a number: '))
while number > 0:
    print(number)
    number -= 1
print('done.')
```

- Once the two statements in the while-loop are executed, the condition will be checked again.
- If the condition does not match, the loop ends, executing the last line in the code.

Nested loop

- Loops can be nested to perform more complicated tasks.
- For example, the following code performs prime factorization of an input:

```
n = int(input())
while n > 1:
    i = 2
    while n % i != 0:
        i += 1
    n //= i
    print(i, end=' ')
print()
```

- Try it and understand how it works.

Exerises (2)

Exercise 2 (1)

- Write a program that reads an integer, then output each of the digits from right to left.
- For example, if the input is `3756`, the program should produce the output of:

```
6
5
7
3
```

- Use a while-loop to complete this task.
- Hint: use modulo operator `%` and floor division operator `//`.

Exercise 2 (2)

- Sample input/output as follow:

Input	Output
1	1
37	7 3
3756	6 5 7 3

for-loop

for-loop (1)

- **For-loop** can be used to iterate through a list of values. A list of values can be generated using the `range()` function.

```
sum = 0
for i in range(10):
    sum += i
print('Sum from 0 to 9 is', sum)
```

- Needless to say, judging from indentation, there is only one statement in the for-loop above.

for-loop (2)

```
sum = 0
for i in range(10):
    sum += i
print('Sum from 0 to 9 is', sum)
```

- `range(10)` will generate a list of values from `0` up to but not including `10`.
- `for i in ...` specify that in each iteration of the loop, variable `i` will be used to keep the current value. Therefore, when `sum += i` is executed the first time, `i` equals `0`.

for-loop (3)

```
sum = 0
for i in range(10):
    sum += i
print('Sum from 0 to 9 is', sum)
```

- The statement `sum += i` will therefore be executed 10 times, each with a different value of `i`.

range()

range(n)

- Now, let's talk about `range()` again.
- `range(n)` with one argument `n` generates a list of integer starting from `0`, up to but not including `n`.

```
sum = 0
for i in range( 10 ):
    sum += i
print('Sum from 0 to 9 is', sum)
```

Output:

```
Sum from 0 to 9 is 45
```

range(start, end)

- If you need a range **starting** from a specific value, you can specify start and stop values instead.
- Start value is inclusive but end value is exclusive.
- So `range(1, 10)` will generate values `1` to `9`. For example:

```
sum = 0
for i in range(5, 10):
    sum += i
print('Sum from 5 to 9 is', sum)
```

Output:

```
Sum from 5 to 9 is 35
```

range(start, end, step)

- The optional third parameter of range can be added to specify the **step** between consecutive values in the sequence.
- So `range(1,10,2)` will generate a list of odd numbers from `1` to `9`.

```
sum = 0
for i in range(1, 10, 2):
    sum += i
print('Sum of all odd numbers from 1 to 9 is', sum)
```

Output:

```
Sum of all odd numbers from 1 to 9 is 25
```

negative range

- All parameter in `range()` can be negative. For example:

```
for i in range(-1, -5, -1):
    print(i)  # output -1, -2, -3, and -4
```

Exerises (3)

Exercise 3 (1)

- Write a program that reads one integer as `n`, then print the first `n` fibonacci numbers.
- First two Fibonacci numbers are `0` and `1`, and the third onwards equal to the sum of the two previous number.
- For example if the input is `5`, the output should be `0 1 1 2 3`.

Exercise 3 (2)

- Sample input/output as follow:

Input	Output
5	0 1 1 2 3
7	0 1 1 2 3 5 8
10	0 1 1 2 3 5 8 13 21 34

Exercises 4 (1)

- Write a program that reads one integer as `n`, then print all prime numbers less than or equal to `n`.
- For example if the input is `5`, the output should be `2 3 5`.

Exercise 4 (2)

- Sample input/output as follow:

Input	Output
5	2 3 5
10	2 3 5 7
25	2 3 5 7 11 13 17 19 23

Advanced topics

- Remember: you may skip this section and come back later.

Advanced loop control

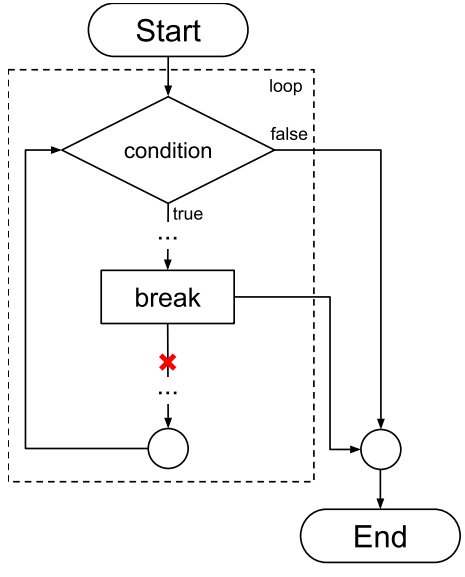
Break and continue

- Sometimes we need more control of how a loop is executed, we may consider using `break` and `continue` in a loop.
- Note that some programmers consider the use of them bad practices, this part is introduced for completeness.

Break

- `break` can be used to leave a loop.

```
i = 0
while True:
    print(i)
    if i > 5:
        break
    i += 1;
```

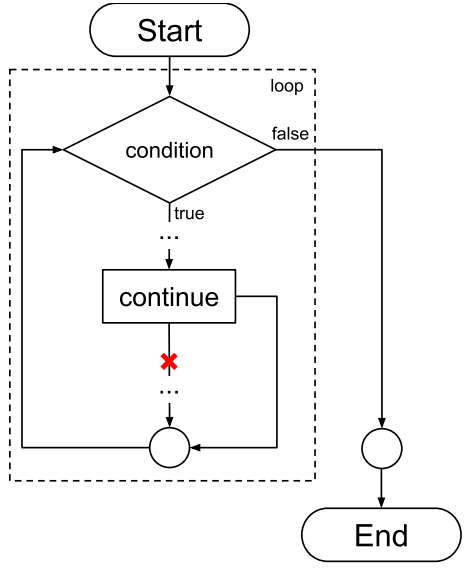


In the above code, when `i` is greater than 5, `break` will end the loop. Try it to see the corresponding output.

Continue

- `continue` can be used to skip the remaining part of a loop.

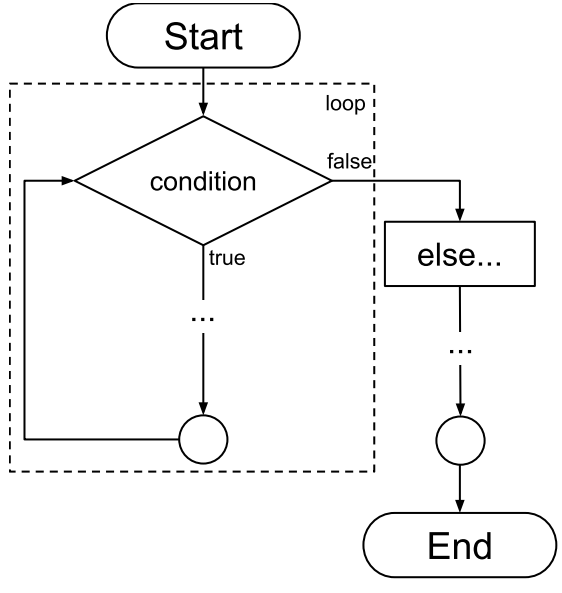
```
i = 0
while i<10:
    if i % 2 == 0:
        i += 1
        continue
    print(i)
    i += 1
```



In this case, `continue` will skip the remaining of the loop, not printing anything for that interation.

Else (1)

- Similar to the case in if-statement, `else` block of a loop will be executed when the loop condition returns `False`.

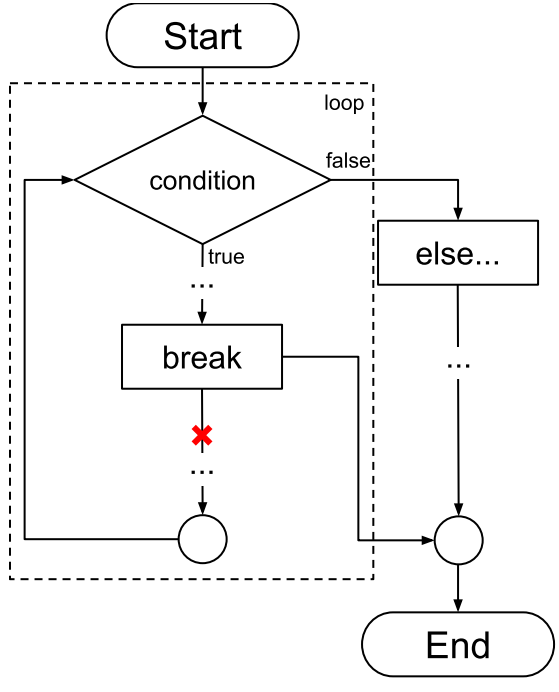



```
i = 1
while i < 10:
    if i % 2 == 0:
        break
    i += 2
else:
    print('Loop ended with i equals', i)
```

Else (2)

- Note that `else` will not be executed if a loop is ended by `break`.

```
i = 0
while i < 10:
    if i % 2 == 0:
        break
    i += 2
else:
    print('Loop ended with i equals', i)
```



- There will not be any output for the code above, as `break` is executed to break the loop.

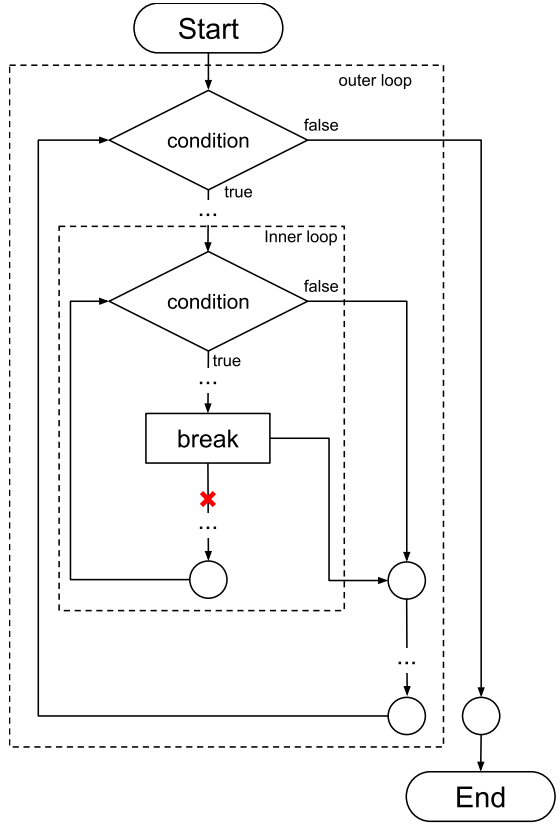
Else (3)

- Note that `else` of a loop does not exist in many other programming languages.

Nested loop

- In a nested loop (loop in a loop), `break` will only break its own loop.

```
for p in range(4):
    for q in range(4):
        if p + q > 1:
            print(p, q)
            break
```



Output:

```
0 2
1 1
2 0
3 0
```

Nested loop (2)

- It is possible to combine the use of `else` and `continue` to break two levels of loop.

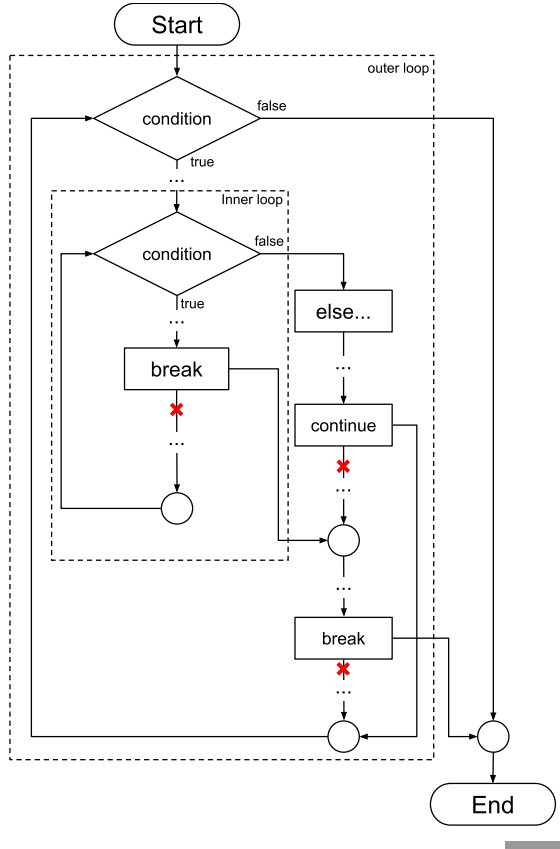
```
for p in range(4):
    for q in range(4):
        if p + q > 1:
            print(p, q)
            break
    else:
        continue
    break
```

- What happens if the `break` in the inner loop is executed?

Nested loop (2) - illustration

- Here is the corresponding flow chart.

```
for p in range(4):
    for q in range(4):
        if p + q > 1:
            print(p, q)
            break
    else:
        continue
    break
```



Nested loop (3)

```
for p in range(4):
    for q in range(4):
        if p + q > 1:
            print(p, q)
            break
    else:
        continue
    break
```

- When `break` in the inner loop is not executed, `else` will be executed.
- If `else` is exectued, `continue` is executed, the final `break` will be skipped.

Nested loop (4)

```
for p in range(4):
    for q in range(4):
        if p + q > 1:
            print(p, q)
            break
    else:
        continue
    break
```

- When `break` in the inner loop is executed, `else` will not be executed.
- If `else` is not exectued, the final `break` will be executed and ends the loop.