CMPUT 274

# Python Control Structures

Topics Covered:

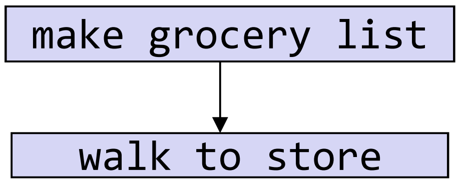
- if/elif/else
- for loop
- range()
- while loop

## Decision Making:
## if/elif/else

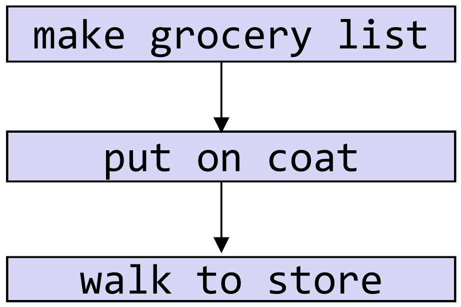
# Decision Making: Motivation

- In many situations, we may want to take a specific action only if a condition is True
- Example: walk to store; if colder than 15C, wear a coat
- Don't want to write 2 different programs: not cold, cold

Program 1:
temperature >= 15C

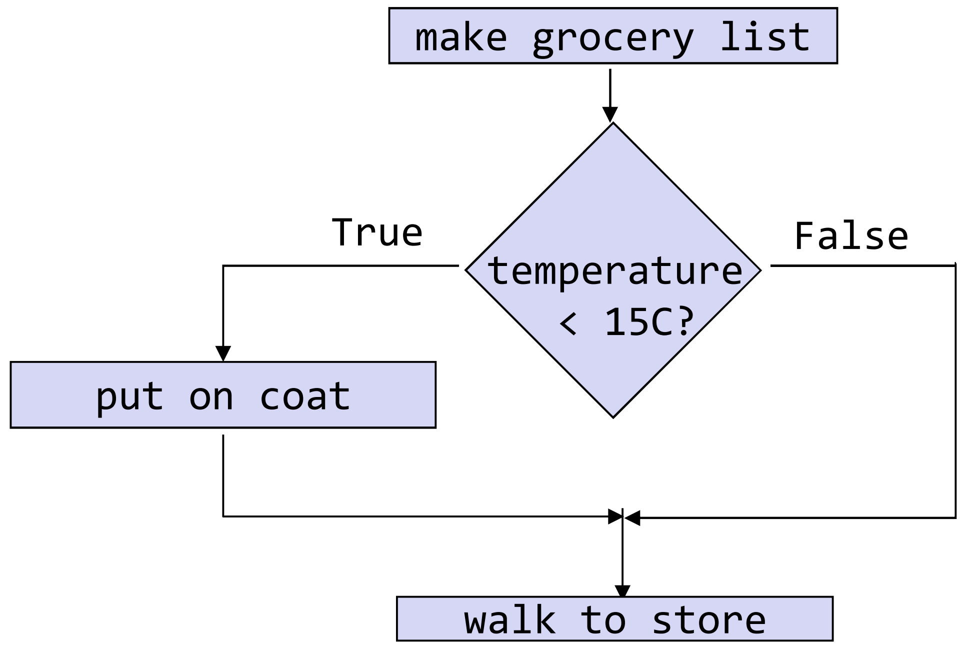


Program 2:
temperature < 15C



# Decision Making: Pseudocode

- Instead, have our program select appropriate path



# Decision Making: Python

```
# ================================
# Buying groceries
# ================================

temperature = int(input("Enter temperature outside in Celsius: "))
print("Making grocery list...")
if temperature < 15:
    print("Putting on coat...")
print("Walking to store...")
```

**SAMPLE RUN 1**

```
Enter temperature outside in Celsius: 20
Making grocery list...
Walking to store...
```

**SAMPLE RUN 2**

```
Enter temperature outside in Celsius: 10
Making grocery list...
Putting on coat...
Walking to store...
```

## `if` Statement

- The `if` statement creates a decision structure, allowing a program to have more than one path of execution.
- It causes one or more statements to execute only when a boolean expression is True.

*BooleanExpression* can be anything that evaluates to True or False

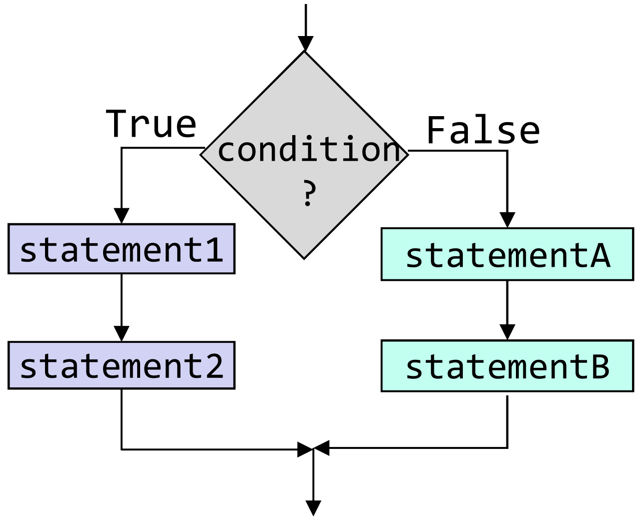
```
if BooleanExpression:
    statement_1
    statement_2
    .
    .
    .
    statement_N
```

Notice **colon** after *BooleanExpression*

All statements that are to be executed when *BooleanExpression* is **True** must be **indented one level**

# `if/else` Statement

- Adds the ability to conditionally execute code when the `if` condition is False



```
if BooleanExpression:
    statement_or_block_if_True
else:
    statement_or_block_if_False
```

Notice **colon** after keyword **else**

# Dual Branch Example

```
# code segment (part of larger program)
# to avoid dividing by zero

if denom != 0:
    ans = num / denom
else:
    print("Cannot divide by zero!")
```

# `if/elif/else` Statement

- Tests a series of conditions (i.e. Boolean expressions)

```
if condition1:
    statement(s)
elif condition2:
    statement(s)
elif condition3:
    statement(s)
⋮
else:
    statement(s)
```

Notice **colon**

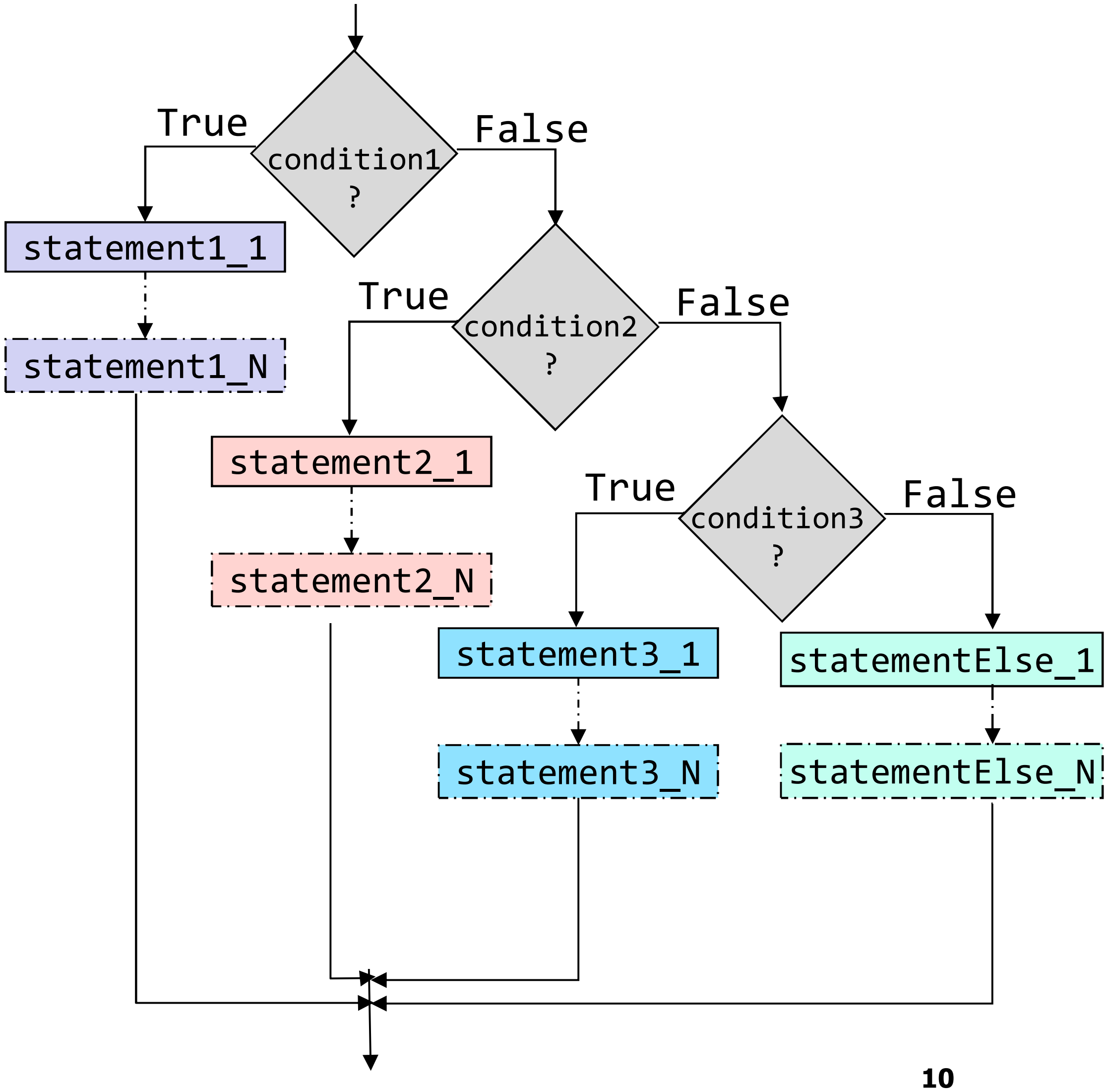
Executes if conditions 1 & 2 are **False**, and condition 3 is **True**

Executes if all conditions are **False**

## `if/elif/else` Statement

- Tests a series of conditions (i.e. Boolean expressions)

```
if condition1:
   statement(s)
elif condition2:
   statement(s)
elif condition3:
   statement(s)
else:
   statement(s)
```



## Multi-branch Example

```
# code segment (part of larger program)
# to determine letter grade

if grade >= 90:
    print("A grade")
elif grade >=80:
    print("B grade")
elif grade >=70:
    print("C grade")
elif grade >= 65:
    print("D grade")
else: print("Failing grade")
```

## Nested `if` Statements

- The statement that is executed under an if, elif, or else can be another if statement

  → nested if statements

```
# code segment: find smallest of 3 values
if num1 < num2:
    if num1 < num3:
        min_val = num1
    else:
        min_val = num3
else:
    if num2 < num3:
        min_val = num2
    else: # if num3 is smallest OR if 3 values equal
        min_val = num3
```

## Same example, no nesting

- May be able to combine nested if statements by combining conditions into one Boolean expression

```
# code segment: find smallest of 3 values
if num1 < num2 and num1 < num3:
    min_val = num1
elif num2 < num3:
    min_val = num2
else: # if num3 is smallest OR if 3 values equal
    min_val = num3
```

## Comparing Decimal Numbers

- Define precision when you say that two decimal numbers are essentially equal (for your purposes)

- Example:

```
# code segment: check if decimal numbers are
equivalent
TOLERANCE = 0.0001
if abs(float_val1 - float_val2) < TOLERANCE:
    print('Essentially equal')
```

## Repetition:
## loops

# Repetition

- What if we want to perform the same action, multiple times?
  - Example: print "Hello World!" 4 times

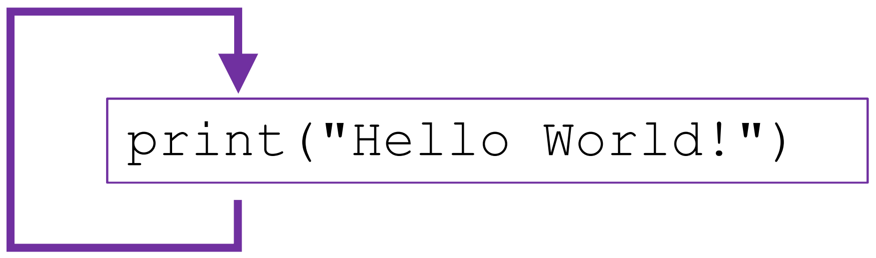
```
print("Hello World!")
print("Hello World!")
print("Hello World!")
print("Hello World!")
```

- Disadvantages of repeating adjacent lines of code:
  - Time consuming
  - Program quickly becomes long
  - Any changes need to be made in multiple places
  - High chance of introducing error

# Loops

- A loop is a way to repeat an action (or set of actions) without writing the same code over & over
  - Example:

Loop 4 times



```
print("Hello World!")
```

- Two types of loops in Python:
  - for loop → when you know how many times to repeat code: *count controlled*
  - while loop → when you <u>don't</u> know how many times to repeat code: *condition controlled*

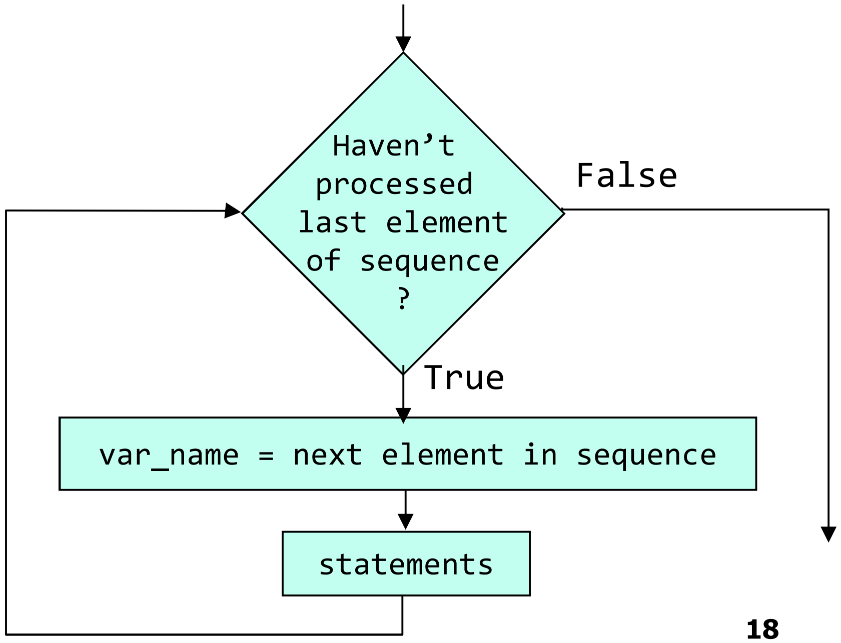
# `for` Loop

- `for` loop repeats code for every element in a given sequence

```
for var_name in sequence:
    statement_1
    statement_2
    •
    •
    •
    statement_N
```

Notice **colon**

All statements that are to be repeated must be **indented one level.**


Haven't processed last element of sequence ?
False
True
var_name = next element in sequence
statements

# `for` Loop: Example

```python
# print words in list
for word in ["1st", "2nd", "3rd"]:
    print(word)
```

```
1st
2nd
3rd
```

- Iteration 1: `word = "1st"`
- Iteration 2: `word = "2nd"`
- Iteration 3: `word = "3rd"`

# for Loop: sequence types

- strings

```
# print characters come before s/S alphabetically
for character in "CMPUT":
    if character.lower()< "s":
        print(character, end="*")
```

```
C*M*P*
```

- lists

```
# calculate sum of elements in a list
total = 0
for num in [10, -2, 3, 24]:
    total += num
print("Sum of elements is", total)
```

```
Sum of elements is 35
```

- tuples, sets

## `range()`

- Can create an immutable sequence of integers using `range()` function

  → returns a range object

Default start is 0

Default step is 1

```
range(start, stop, step)
```

Up to, but not including, **stop**

- Must always include stop value; start and step are optional

## `for` Loop with `range()`

- Specify stop value only

```
for num in range(5):
    print(num, end=', ')
```

```
0, 1, 2, 3, 4,
```

- Specify start, stop, step

```
for odd_num in range(3,11,2):
    print(odd_num, end=', ')
```

```
3, 5, 7, 9,
```

- Count backwards

```
for num in range(5, 0, -1):
    print(str(num) + '...')
print('Blast off!')
```

```
5...
4...
3...
2...
1...
Blast off!
```

## Use `range()` with another sequence

- Recall:

```
# calculate sum of elements in a list
total = 0; my_list = [10, -2, 3, 24]
for num in my_list:
    total += num
print("Sum of elements is", total)
```

```
Sum of elements is 35
```

- Can also use range of indices to traverse list:

```
# calculate sum of elements in a list
total = 0
my_list = [10, -2, 3, 24]
for i in range(len(my_list)):
    total += my_list[i]
print("Sum of elements is", total)
```

```
Sum of elements is 35
```

# `while` Loop

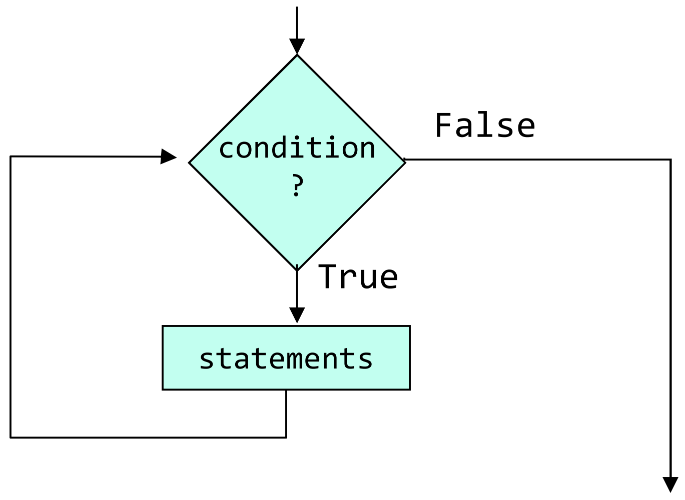
- `while` loop repeats code as long as a condition is <u>True</u>

*condition* can be anything that evaluates to **True** or **False**

```
while condition:
    statement_1
    statement_2
    •
    •
    •
    statement_N
```

Notice **colon**

All statements that are to be repeated must be **indented one level**



## `while` Loop: Example 1

- Repeat an action until the user enters a specific value: a sentinel value

```
# add values entered by user, until wants to stop
SENTINEL = 0
total = 0
num = -1  # initialize -> different from sentinel
while num != SENTINEL:
    num = int(input("Enter value to add; 0 to stop"))
    total += num
print("Sum of values entered by user is", total)
```

- Sentinel value chosen must be appropriate for problem. e.g. What would be a good choice if multiplying values above instead of adding?

# `while` Loop: Example 2

- Can also use a `while` loop to traverse a sequence of known length (instead of using a `for` loop)

  → if in doubt, can always used `while` loop

```
# calculate sum of elements in a list
total = 0
my_list = [10, -2, 3, 24]
i = 0  # INITIALIZE control variable
while i < len(my_list):
    total += my_list[i]
    i += 1  # UPDATE control variable
print("Sum of elements is", total)
```

```
Sum of elements is 35
```

## Beware: Infinite Loops

- Something inside the loop should eventually make the `while` condition False
  - → Otherwise, the loop will continue to repeat forever (or program is manually terminated)

- So at least one thing related to the condition expression must be updated every iteration

# `break`

- Can use break to exit from a loop (even an infinite loop)
  - → demonstrated in Python Intro Labs (page 59)
- While learning, avoid break: better to use flags
  - → code easier to read

```
correct_answer = False
while not(correct_answer):
    name = input("Guess name of this course: ")
    if name == "CMPUT 274":
        print("Correct!")
        correct_answer = True
    else:
        print("Incorrect. Try again!")
```

# Nested Loops

- The code that is repeated inside a loop can be whatever we choose, including <u>another</u> loop

  → nested loop

```
for row in range(3):
    print(row, end=": ")
    for col in range(5):
        print(col, end=" ")
    print()
```

```
0:  0 1 2 3 4
1:  0 1 2 3 4
2:  0 1 2 3 4
```