IT@JCU

CP1401 – Module 3
Decision Structures

JAMES COOK
UNIVERSITY
AUSTRALIA

Learning outcomes – you will be able to:

- Test programs with meaningful test values
- Choose appropriate decision structures
- Use Boolean logic to create compound conditions
- Understand boundary conditions and check for valid or invalid situations
- Write Python code to implement programs using decision structures

Do this now - write an algorithm for:

Get:

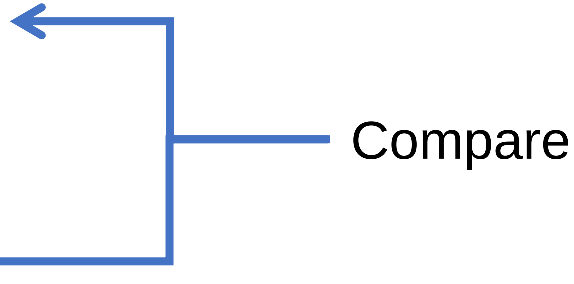
- the size of a block of land,
- the price per square metre of land,
- and the price of a house.

Calculate and display the total cost of the package.

Testing is an important part of problem solving

Evaluate your algorithms *before* coding them

- What SHOULD it do?



Compare

- What DOES it do?

- If they don’t match, then the algorithm is wrong…
  Don't write code to implement an incorrect algorithm!
- Step through it, line by line (or step by step) using test values, "known-good" results

How do we know if this algorithm is correct?

```
get land_size, land_cost and house_cost
total_land_cost = land_cost + house_cost
package_cost = total_land_cost * land_size
```

- Test it with **meaningful** inputs (known-good or easy to calculate)

  land_cost = 5

  land_size = 100

  house_cost = 1000

- package_cost SHOULD be 1500… *is it?*

Correct and improve your algorithms

- If it **doesn’t** work:
  - Why? How do we fix it?
    - Are there any missing/incorrect steps?
    - Are any steps in the wrong place?
    - Typos? Inconsistencies?

- If it **does** work:
  - Can we improve it?
    - Are there any unnecessary steps?
    - Can we make it more **efficient** or more **readable**?

Corrected algorithm… or is it?

```
get land_size, land_cost and house_cost
total_land_cost = land_cost * land_size
package_cost = total_land_cost + house_cost
```

- Be systematic! Test it again with the same values.

    land_cost = 5

    land_size = 100

    house_cost = 1000

- package_cost SHOULD be 1500… *is it?*

Let's "desk check" these algorithms

Determine if you can afford to buy an item based on your money and the item price

```
get your_money, item_price
if your_money >= item_price
    buy it
else
   can't buy it
```

Let's "desk check" these algorithms

Calculate speed in kilometres per hour given distance in km and time in minutes

```
get distance_in_kilometer, time_in_minutes
speed = distance_in_kilometer / time_in_minutes
print speed
```

Let's "desk check" these algorithms

Calculate speed in kilometres per hour given distance in km and time in minutes

```
get distance_kilometer, time_minutes
speed_kilometer_per_hours = distance_ kilometer / time_minutes
print speed_kilometer_per_hours
```

Test your code

- Why?
  - We need to check that we have implemented our solution correctly

- How?
  - Try running it, see what happens ☺
  - If and when error messages appear, READ THEM!
  - Right now they may seem unclear, but we will get more familiar with them

What are we testing?

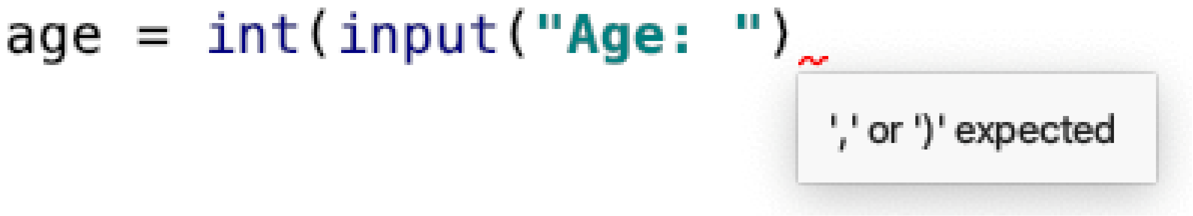
- That it **runs**
  - No syntax errors
- That it gives the **correct** results
  - No logic errors

- Are there any user inputs that break the program?
  - No runtime errors… This is not a concern for us yet.

Syntax errors will happen

- How do we identify them?
  - Run code
- How do we find them?
  - READ error messages… No, really, READ them!
- How do we fix them?
  - With practice and familiarity - get used to common errors and their fixes

- Let PyCharm help you find and fix syntax errors



```
age = int(input("Age: ")
```

',' or ')' expected

Read and understand your error messages

```
File "/demos/ipo.py", line 6
    monthly_cost = float(input"Monthly cost: $")
                                               ^
SyntaxError: invalid syntax

Traceback (most recent call last):
  File "/demos/ipo.py", line 8, in <module>
    pint(total)
NameError: name 'pint' is not defined
```

Logic errors will happen

- How do we identify them?
    - Test systematically
    - Incorrect results indicate logic errors
    - What makes good test data?
        - KNOWN results (otherwise what are you comparing it to?)
        - A "representative sample" of data that covers the possibilities
- How do we locate them?
    - Debugging
- How do we fix them?
    - Modify code and maybe algorithm

Syntax vs logic errors

- Syntax errors are (usually) easy to find
  - Python will tell you which line (or close) is wrong!
- Logic errors can be sneaky
  - So we need to test
  - Modern programming environments provide debuggers to help trace code
- Both are corrected by modifying code
  - Either to fix the syntax, or to use the correct logic

Decision Structures

Do this now

- Write an algorithm to help a bouncer decide if a guest is allowed into a nightclub. The bouncer should ask the guest for their age and then tell the guest either "allowed" or "denied" depending on if they are old enough (18+) to enter.

- Write the Python code for this *after* you have written the algorithm.

Look for decisions during problem decomposition

- What words in a problem description indicate decisions?
    - Look for words like **if**, **then**, **otherwise**, **else** and so on
        - SOME of these will be handled using decisions
        - OTHERS may be handled with repetitions (loops)
    - Read the description carefully
        - **if** a decision is implied in the wording **then** it is probably needed in the solution

Use decomposition to look for decisions in this:

- Our program asks the user to enter a number. If the number is less than zero then the message “Invalid number” is displayed. Otherwise, the message “All good” is displayed.

- No repetition here, so this is a decision

Let's help our bouncer

- Write an algorithm to help a bouncer decide if a guest is allowed into a nightclub. The bouncer should ask the guest for their age and then tell the guest either "allowed" or "denied" depending on if they are old enough (18+) to enter.

```
get age
if age >= 18
    print "allowed"
else
    print "denied"
```

Now let's convert the algorithm to Python code

```
get age
if age >= 18
    display "allowed"
else
    display "denied"
```

```python
age = int(input("Age: "))
if age >= 18:
    print("allowed")
else:
    print("denied")
```

All decisions are done with **conditions**

- In both algorithm design and programming,
  all decisions include a **condition**
- **All** conditions are **expressions** that evaluate to either:
  - True
  - False
- That means, all conditions are…?
  - **Boolean expressions**

Relational operators compare values

Expression	Explanation
`x == y`	x is equal to y
`x != y`	x is not equal to y
`x < y`	x is less than y
`x > y`	x is greater than y
`x <= y`	x is less than or equal to y
`x >= y`	x is greater than or equal to y

Learn these by practising with them in the Python console (now)

Strings can be compared using the same operators

- String comparisons are case-sensitive
- Compared character-by-character based on the ASCII values
- If shorter word is substring of longer word, longer word is greater than shorter word

Expression	Result
`"Hello" == "hello"`	False
`"Help" > "Hell"`	True
`"Help" >= "Help"`	True

Again, 'play' with this in the console until it makes sense

Relational operators are the same in most languages

Pseudocode:

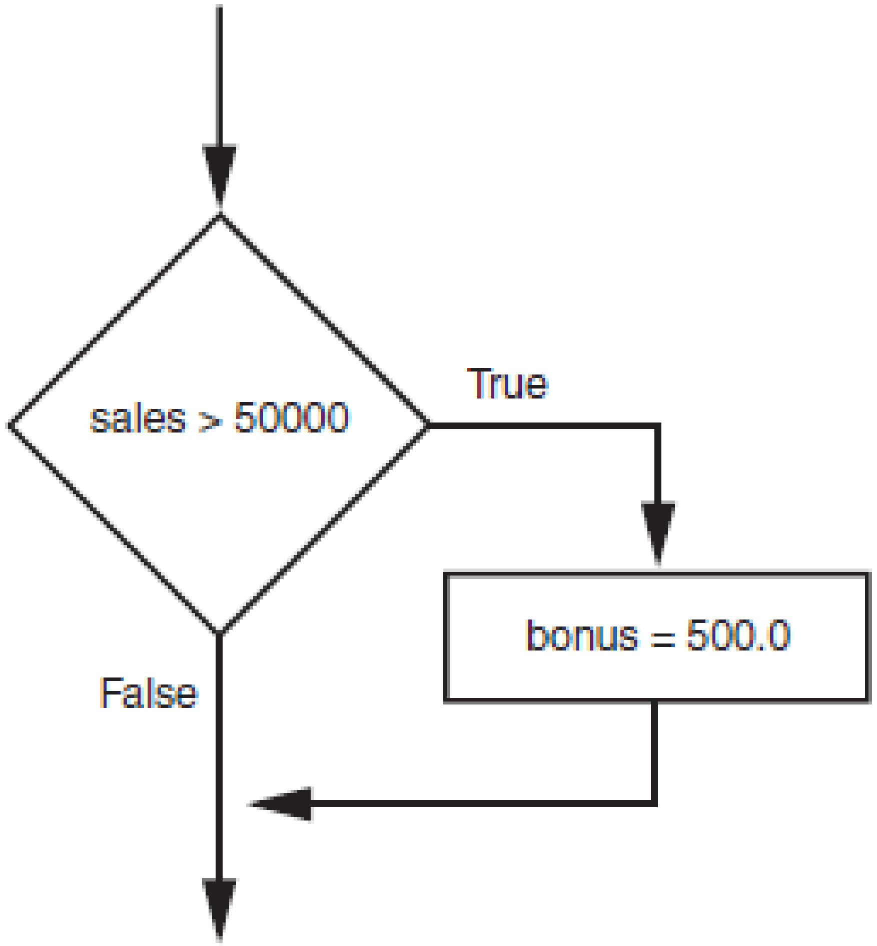
```
if sales > 50000
    bonus = 500.0
```

Python:

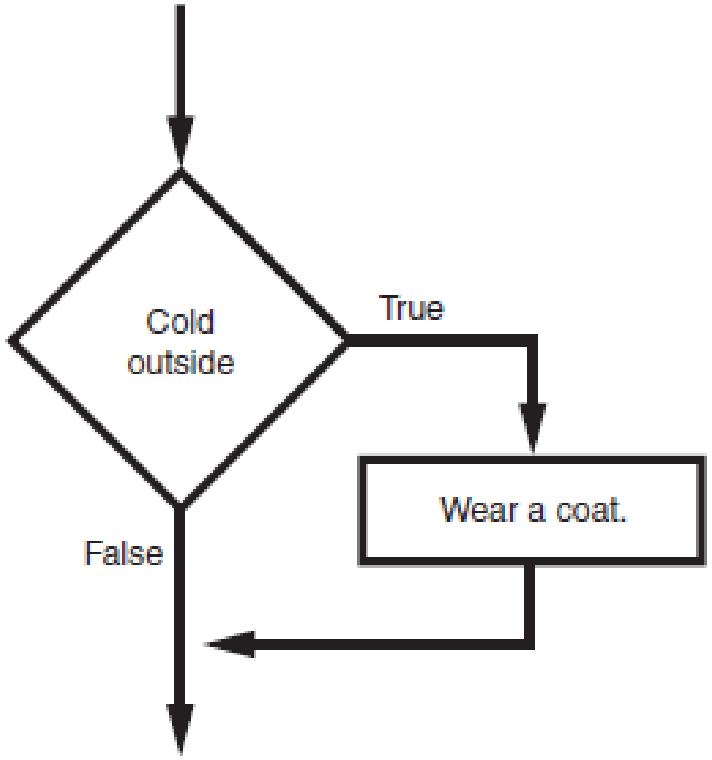
```python
if sales > 50000:
    bonus = 500.0
```

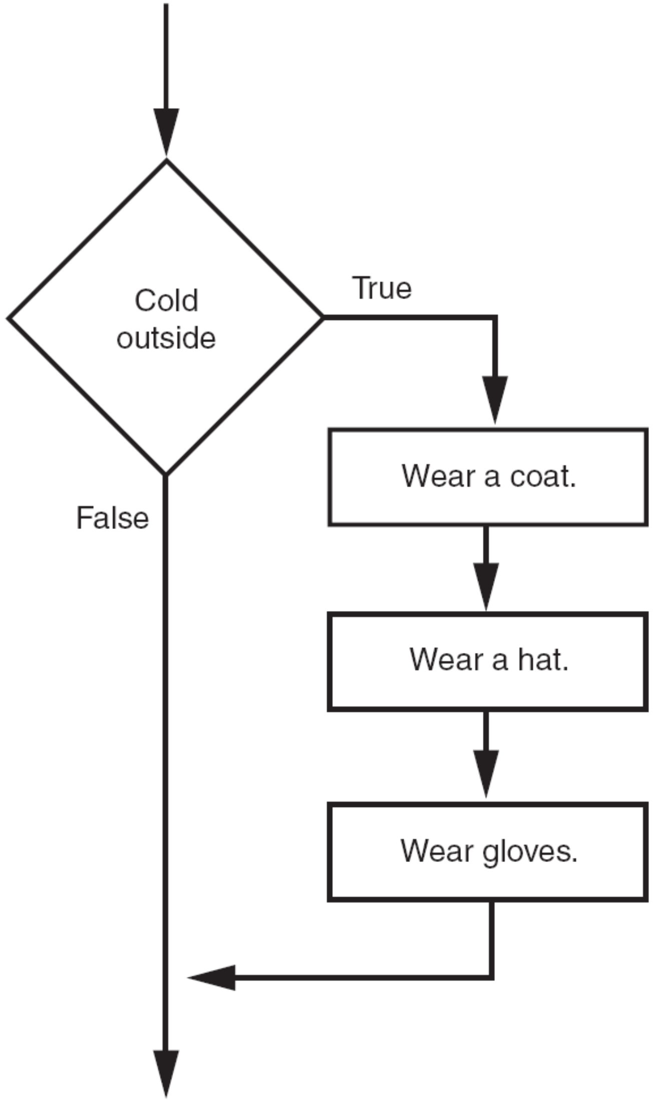
Java and C/C++:

```
if (sales > 50000)
    bonus = 500.0;
```



Use diamonds for decisions in flowcharts





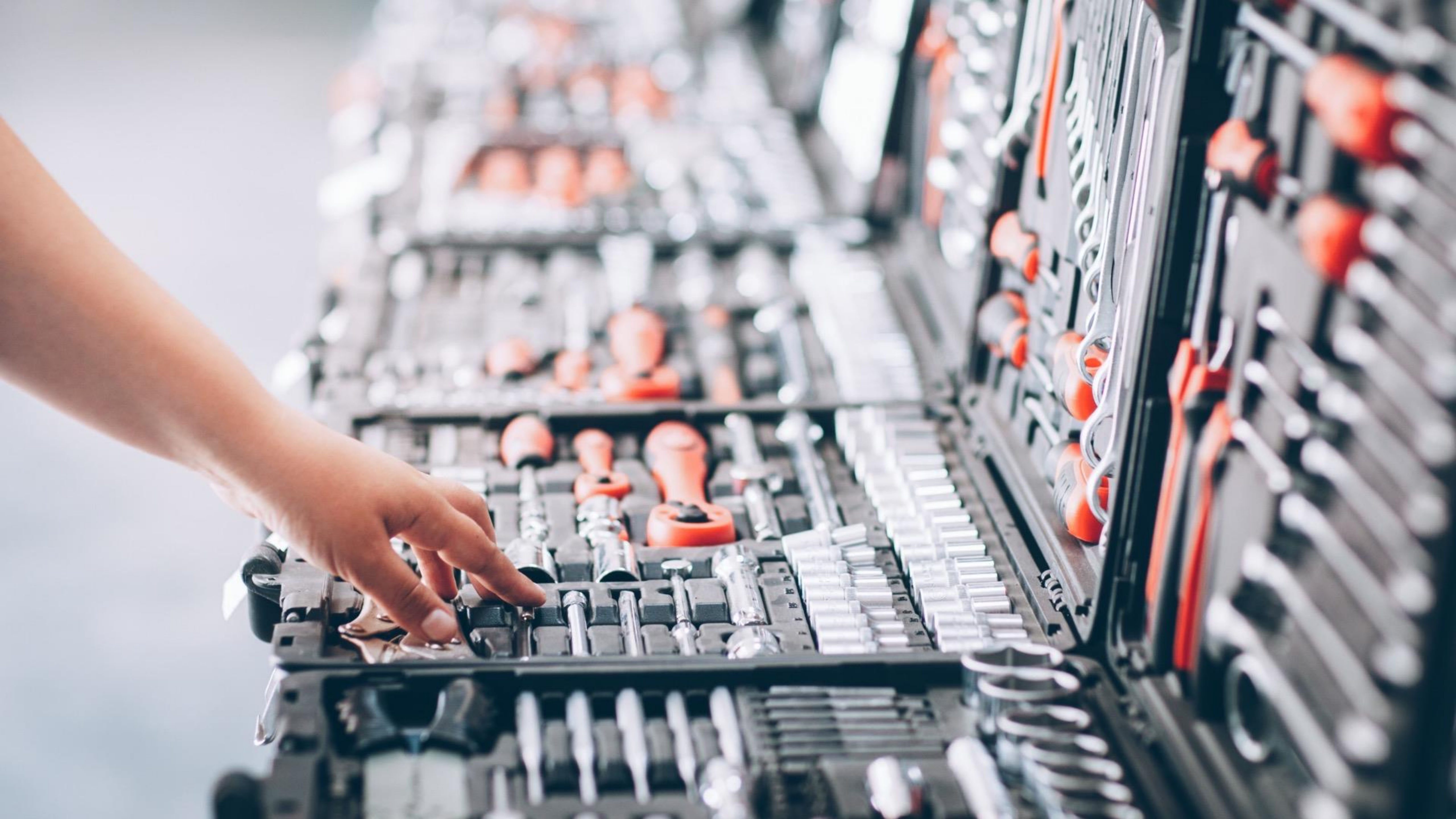
Deconstructing the if statement in Python

```
if condition:
    statement
    statement
```

- First line is known as the "if clause", and includes the keyword `if` followed by a condition, followed by a colon
    - The condition can (only) be True or False
    - When the if statement executes, the condition is tested, and if it is True, the block statements (indented part) are executed.
      Otherwise, block statements are skipped.

There are 5 common decision patterns

Learn these and when to use them



Look for ‘patterns’ of problems/solutions

- Start getting familiar with how certain problems are solved
- Similar problems usually have similar solutions: patterns
  - Avoid reinventing the wheel, just modify it for our specific case
- Learn and practise these patterns

We've already seen simple if statements with no else

Example:

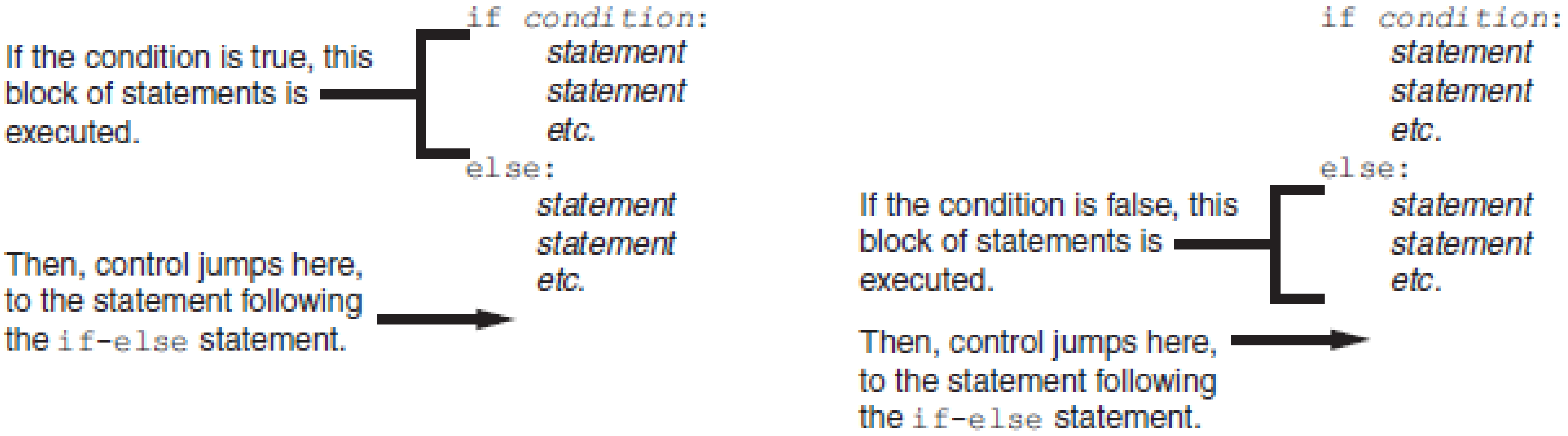
```
if age >= 100:
    print("Wow, that's amazing!")
print("Your age is", age)
```

Use this pattern when you don't want to do anything different if the condition is False.

if, else statements have two paths (True, False)

- If the condition is True, the False path (else) statements are skipped
- If the condition is False, the True path (if) statements are skipped

Figure 4-7 Conditional execution in an `if-else` statement



```
if condition:
    statement
    statement
    etc.
else:
    statement
    statement
    etc.
```

If the condition is true, this block of statements is executed.

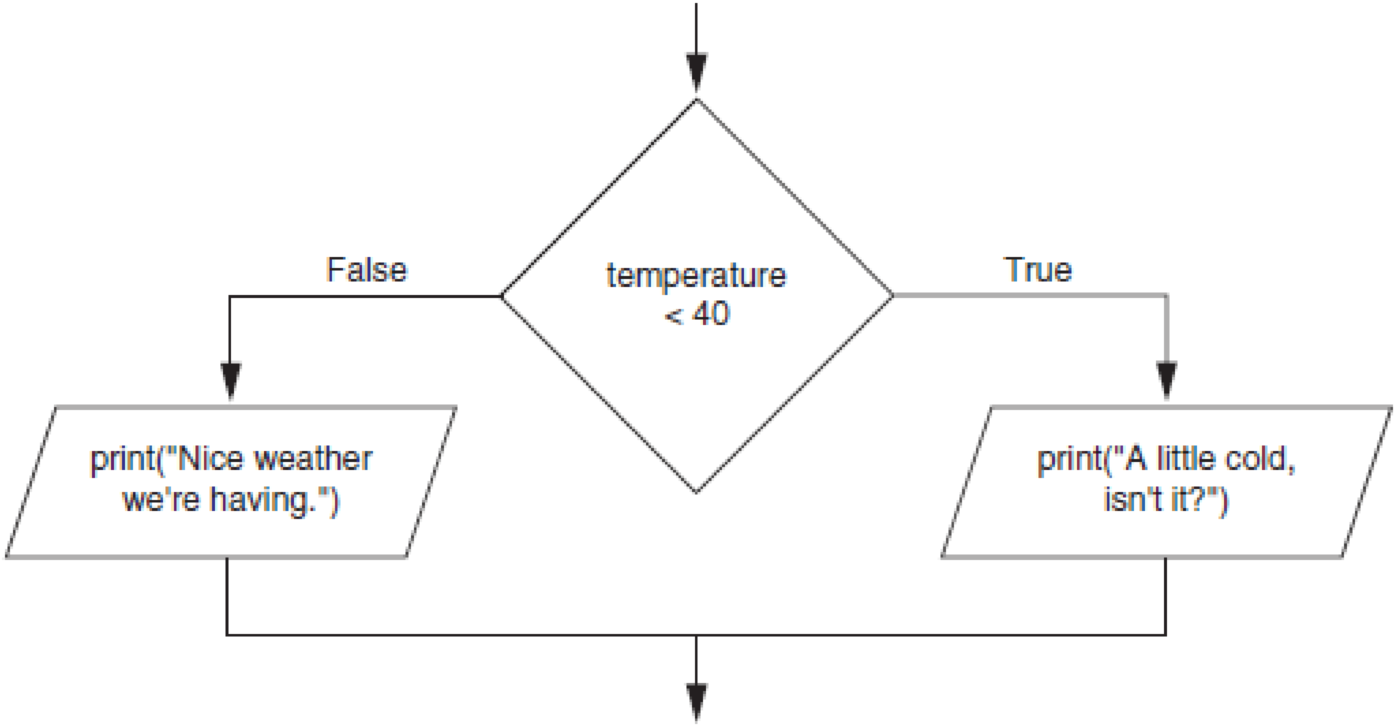
Then, control jumps here, to the statement following the `if-else` statement.

```
if condition:
    statement
    statement
    etc.
else:
    statement
    statement
    etc.
```

If the condition is false, this block of statements is executed.

Then, control jumps here, to the statement following the `if-else` statement.

Do this now - write the code for this flowchart


False
temperature < 40
True
print("Nice weather we're having.")
print("A little cold, isn't it?")

```
if temperature < 40:
    print("A little cold, isn't it?")
else:
    print("Nice weather we're having.")
```

Anti-pattern: Do NOT use elif with a condition when you just need else

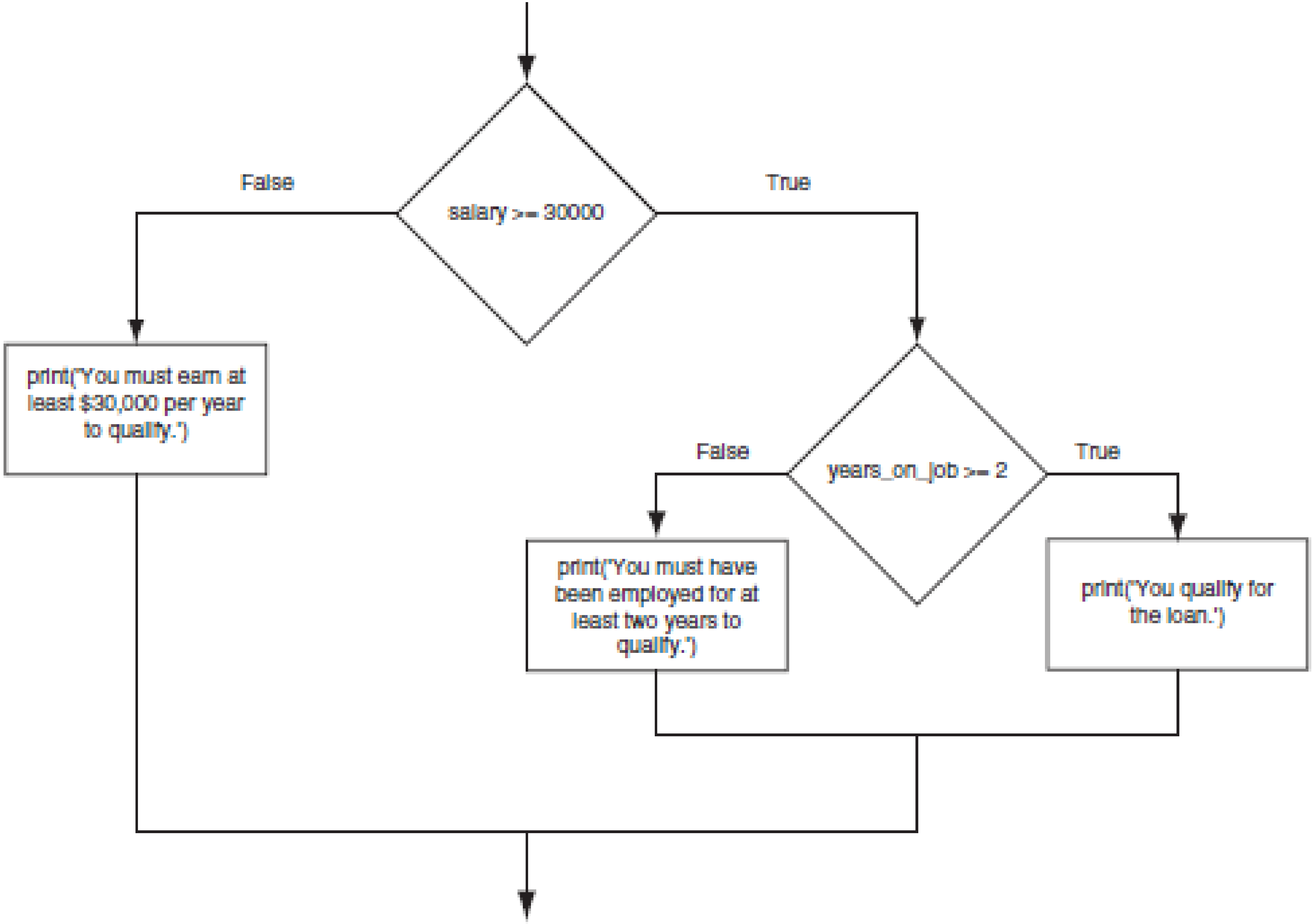
- If you don't need to ask another question, don't ask!
  - You might make a mistake (watch the boundary!)
  - It's less efficient
  - It's harder to read (you need to process it and so does the computer)

```
if temperature < 40:
    print("A little cold, isn't it?")
elif temperature > 40:
    print("Nice weather we're having.")
```

You can nest decision structures to handle more than two paths

- A decision structure can be **nested** inside another decision structure
- Example:
  - To determine if someone qualifies for a loan, they must meet two conditions:
    - Must earn at least $30,000/year
    - Must have been employed for at least two years
  - Check first condition, and if it is True, check second condition

Figure 4-14 A nested decision structure


salary >= 30000
False
True
print('You must earn at least $30,000 per year to qualify.')
years_on_job >= 2
False
True
print('You must have been employed for at least two years to qualify.')
print('You qualify for the loan.')

Code for nested if statement example

```
if salary >= 30000:
    if years_on_job >= 2:
        print("You qualify for the loan.")
    else:
        print("Must be employed for at least 2 years.")
else:
    print("You must earn at least $30,000.")
```

Python requires proper indentation

- Indenting makes a difference to whether your code works!
- Makes code more readable for programmer
    - Good indentation is still best-practice in languages like Java, C, etc. that do not actually require it (they use `{}` to denote blocks) because it's easier to scan well-formatted code.
- Rules for writing nested if statements:
    - `else` clause must align with its matching `if` clause
    - Statements in each block must be consistently indented

Python requires proper indentation

```
if salary >= 30000:
print("Broken")
```

```
IndentationError:
expected an indented
block
```

```
if salary >= 30000:
    if years_on_job >= 2:
        print("You...")
    else:
  print("You..")
```

```
IndentationError:
unindent does not match
any outer indentation
level
```

Do this now

Write Python code for this simple menu-style problem

Print a greeting depending on the user's choice. If the choice is "h", print "Hello", if it is "g", print "Goodbye", if it is "w", print "Wazzzup!", and if it is anything else, print "Whatever".

if, elif, else statements simplify nested decisions

- if-elif-else can make the logic of nested decision structures simpler to read and write. You can have unlimited elif statements.

```
choice = input("Choice: ")
if choice == "h":
    print("Hello")
elif choice == "g":
    print("Goodbye")
elif choice == "w":
    print("Wazzup!")
else:
    print("Whatever")
```

This is Python's equivalent of the **switch** or **case** statement that other languages have.

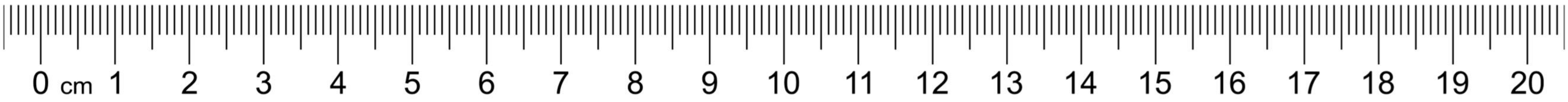
We use this for menus or similar

Be systematic when determining place in a continuum

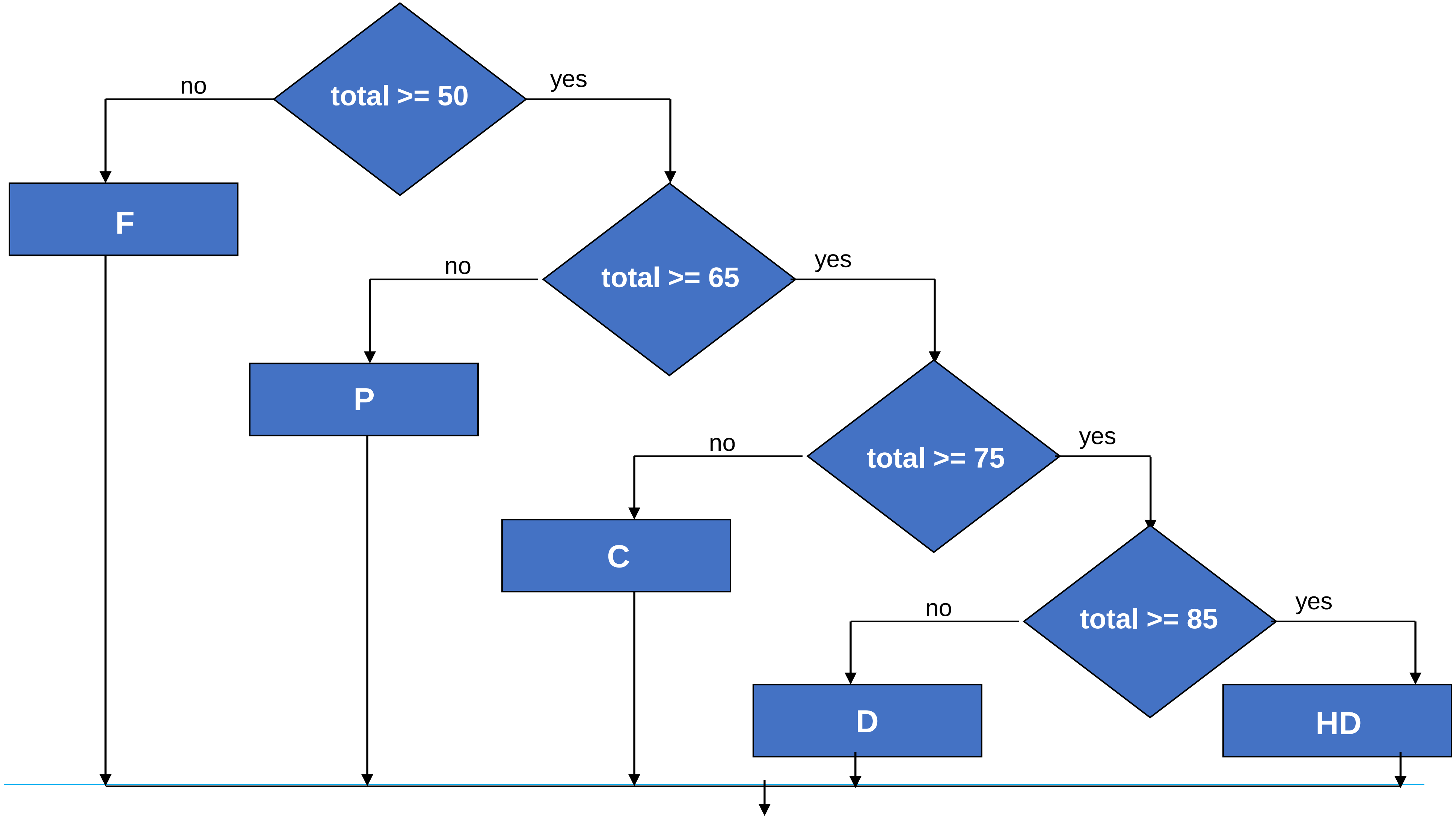
- Start at one end, move in the same direction
- You only need one condition each check

Example: Negative, 0-10, 10-15, 15-20, 20+

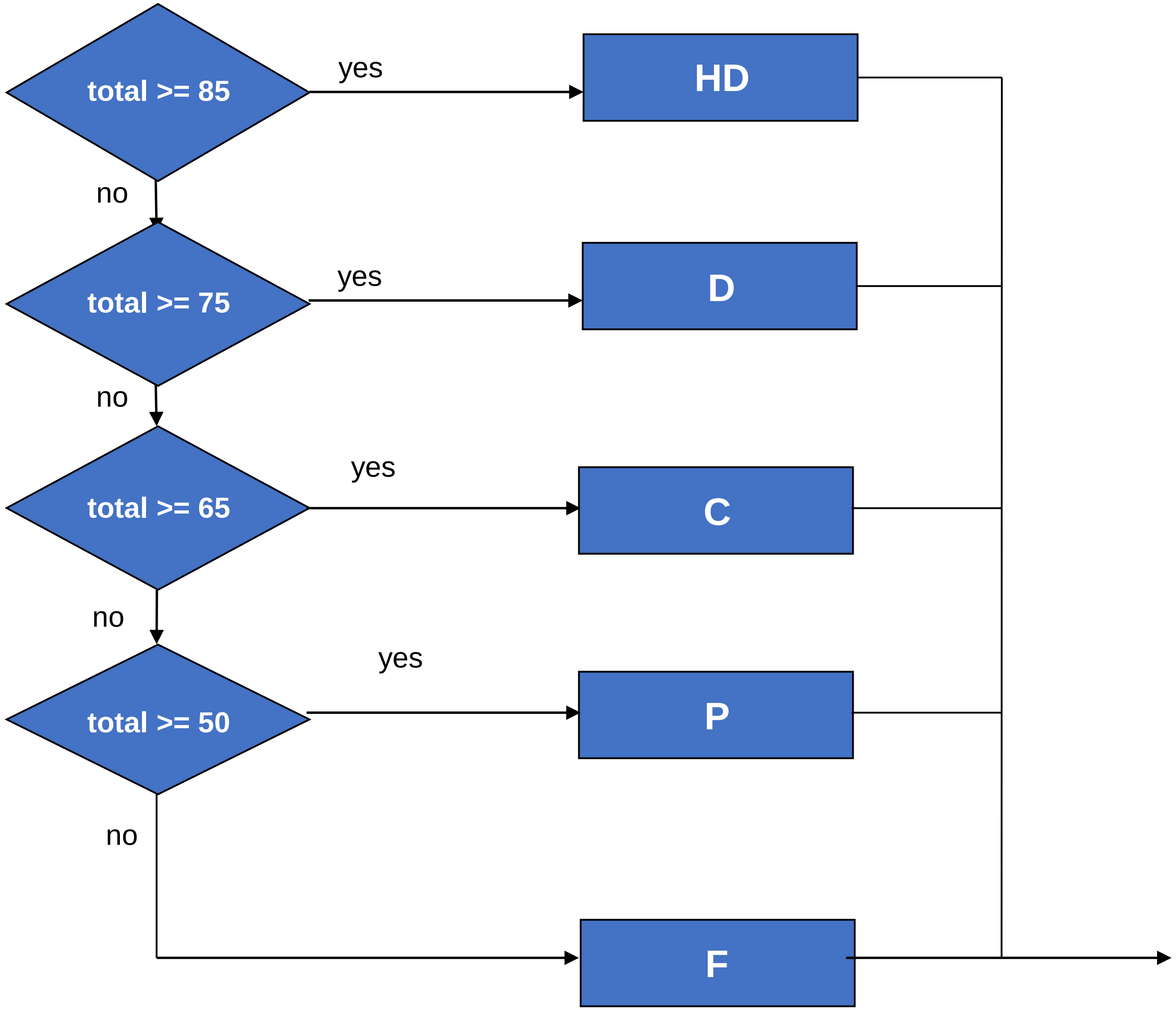
NOT: 10-15, 0-10, 20+, 15-20, Negative



Nested if, else statements version
Determining JCU grade based on total subject result



if, elif, else version



if, elif, no else

- Sometimes we have some special cases but no default/other case
- E.g., here we print a special message for high scores, but nothing gets printed for normal/other scores:
- I.e., there's another case (scores < 80), but no extra path for it

```
if score >= 90:
    print("Your score is very good")
elif score >= 80:
    print("Your score is good")
print("Your score is", score)
```

if, elif, no else is uncommon

- If you use this pattern, ask yourself:
  - "What case(s) do I NOT want to handle?", or
  - "What scenario(s) do I want to ignore?"
- No answer? Don't use this pattern.

```
if score >= 50:
    print("You pass :)")
elif score < 50:
    print("You fail :(")
print("Your score is", score)
```

This should just be else

if, if, if...

- In all our previous patterns, the paths were mutually exclusive - e.g., never old *and* young; hello *and* goodbye *and* wazzzup…
- In some situations, our conditions (and paths) are not mutually exclusive - e.g.,

```
if score >= 50:
    print("You passed")
if score >= 90:
    print("You win a car!")
if score >= 80:
    print("You win a horse :)")
```

Learn when to use each decision pattern

See our patterns page: https://github.com/CP1404/Starter/wiki/Programming-Patterns#decision-structures

Pattern	What it does	Example

if, no else	Do something when the condition is True, but do nothing when it's False	`if score >= 90:`
		`    print("That's exceptional!")`

if, else	Do something when the condition is True, do another (something else) when False	`if score >= 50:`
		`    print("Pass")`
		`else:`
		`    print("Fail")`

if, elif, else	Handle **3 or more mutually exclusive cases**
	You can have any number of elifs,
	and the last else catches anything… else
	Common for menus
		`if score >= 90:`
		`    print("Excellent")`
		`elif score >= 50:`
		`    print("Passable")`
		`else:`
		`    print("Bad")`

if, elif, no else	Handle **multiple mutually exclusive** cases, but with no default path
	(Rare)
		`if score >= 90:`
		`    print("Score is very good")`
		`elif score >= 80:`
		`    print("Score is good")`

if, if, if	Potentially multiple outcomes.
	Handle cases that might overlap – **not mutually exclusive**
		`if score >= 90:`
		`    print("You win a car!")`
		`if score >= 80:`
		`    print("You win a horse :)")`

Do this now

Write code for this partial algorithm:

- Given pool pH level:
  - 7.4 - 7.6 is ideal, no change
  - Below 7.4, is too acidic, add soda
  - Above 7.6 is too alkaline, add acid
- Which pattern will you use?

Boolean logic

Use Boolean (logical) operators to create complex conditions

- There are only 3 operators: **and**, **or**, **not**
- and, or: binary operators (two operands)
- not operator: unary operator, reverses the truth of its one operand

- You can get a loan if you earn > 30K **and**
  you've been working 2 or more years

- You can get into the nightclub if you are over 18 **or**
  you bribe the bouncer

and and **or** can simplify decision structures

- **and** expression is True only when **both** operands are True
- **or** expression is True when **either** operand is True

Truth tables for and and or operators:

Expression	Value
False **and** False	False
False **and** True	False
True **and** False	False
True **and** True	True

Expression	Value
False **or** False	False
False **or** True	True
True **or** False	True
True **or** True	True

The not operator

- Takes one Boolean value as its operand and reverses this value
- Sometimes it may be necessary to place parentheses around an expression to clarify what you are applying the not operator to

```
not (x > y or x < z)
```

Truth table for the not operator:

Expression	Value
not True	False
not False	True

Do this now

Rewrite this code using Boolean operator/s so you have a single if-else with the same outputs/results.
(Notice here there are two "No" paths, which we want to combine.)

```python
if salary >= 30000:
    if years_on_job >= 2:
        print("Yes")
    else:
        print("No")
else:
    print("No")
```

Understand operator precedence to create complex expressions as needed

- Precedence: arithmetic, then relational, then logical/Boolean
  - `a > b + 5 * c and d`
  - `(a > (b + (5 * c))) and d`

add brackets to this now

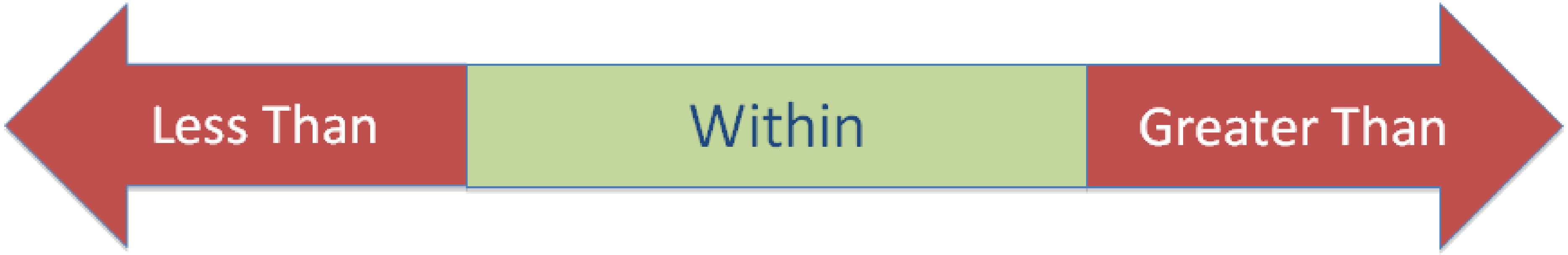
- Any expression with a logical or relational operator will result in a Boolean result

x = 1, y = 2, a = 3, b = 4	Result
`x > y and a < b`	?
`not x > y`	?
`x + y < 3 and b > a`	?

Use logical operators to check numeric ranges (useful for error checking)

- To determine if a value is **within** a range, use the **and** operator and appropriate relational operators
  - E.g., `age >= 0 and age <= 120`

- To determine if a value is **outside** a range, use the **or** operator and the *opposite* relational operators
  - E.g., `age < 0 or age > 120`



You can store the result of a Boolean expression

```
if salary >= 30000 and years_on_job >= 2:
    is_qualified = True
else:
    is_qualified = False

# later...

if is_qualified:
    print("You qualify!")
```

Note: you use Booleans as conditions directly. You do NOT need to compare them to True or False (they are already either True or False)

Since conditions are already True or False...

```
is_qualified = salary >= 30000 and years_on_job >= 2



# later...

if is_qualified:
    print("You qualify!")
```

Boolean variable names should sound like Booleans!

- Names are so (SO) important.
- When you read a name, you should know what it means
- Booleans should read like accurate simple English:

```
if is_qualified:
```

- So, is_something, has_something, will_something… (mostly **is**)

Do this now

Come up with good names for these variables now:

- Whether or not a person is an adult
- If a number is prime
- The roast level of a batch of coffee beans
- Whether a user is an administrator

Test, test, test...

… test systematically

Test your decisions systematically (carefully)

- Test all the possible paths in a decision statement
- After testing all the actions, examine all the conditions
- Rigorous testing includes checking conditions that contain compound Boolean expressions using data that produce all the possible combinations of values of the operands
- Test unexpected values (within reason)
- A representative sample is enough; don't need all possible values.

- Note: we won't handle unexpected *types* in this subject
  E.g., a user entering 'a' when we ask for an integer

What is the minimum number of tests we need for this code?

```
age = int(input("Age: "))
if age > 100:
    print("You are a centenarian!")
else:
    print("You are not very old")
```

Test each path - AND any boundaries

- We need at least **3**:
  - True path
  - False path
  - Boundary

```
age = int(input("Age: "))
if age > 100:
    print("You are a centenarian!")
else:
    print("You are not very old")
```

Use test data with known outcomes, compare to actual outcomes

Input (age)	Expected outcome	Actual outcome
10	`"You are not very old"`	?
111	`"You are a centenarian!"`	?
100	`"You are a centenarian!"`	?

Test every boundary condition

- "Boundary conditions" are a very common source of problems

```
age = int(input("Age: "))
if age > 18:
    print("allowed")
else:
    print("denied")
```

- Looks OK - only allow people in if they are over 18.
- But… is it?

What test values do we need for these programs?

```
if salary >= 30000:
    if years_on_job >= 2:
        print("Yes")
    else:
        print("No")
else:
    print("No")
```

```
if salary >= 30000 and \
   years_on_job >= 2:
    print("Yes")
else:
    print("No")
```

salary	years_on_job	Expected outcome	Actual outcome
30000	2	"Yes"	?
10000	5	"No"	?
10000	1	"No"	?
40000	1	"No"	?
…			

Now do these next steps

- Find an everyday process that uses decisions and rewrite it as an algorithm in pseudocode
- Practise writing algorithms and programs that use decision structures
- Practise testing your algorithms and code systematically
- Read chapter 4 of your textbook (repetition structures)