Computer Programming I

Conditional Branching

Control Structures

- Control structure: A logical design that controls the order in which set of statements execute

1. Sequence structure: A set of statements that execute in the order they appear
2. Decision structure: Specific action(s) performed only if a condition exists
   - Also known as selection structure or conditional branching
3. Repetition Structures: repeat a set of statements as many times as necessary
   - Also known as loops

Decision Structures

- Some problems simply *cannot* be solved by performing a set of ordered steps, one after another (sequence structure)
- For example consider a company payroll program that determines whether an employee has worked overtime
  - If the employee has worked more than 40 hours, he or she gets paid a higher wage for the hours over 40
  - Otherwise, the overtime calculation should be skipped
- Solving this kind of problem requires a decision structure

Boolean Expressions

- Expression that yields a value of `True` or `False`
  - Boolean expressions are also known as conditions
  - Decision structures are based on a condition
- Typically Boolean expressions are formed with relational operators
- Relational operator: determines whether a specific relationship exists between two values
  - Relational operators are also known as Comparison Operators

Relational Operators

- These operators are binary: they take two values
- These operators yield a Boolean value: `True` or `False`

Operator	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

Relational Operators: Examples

Expression	Meaning
`x > y`	Is x greater than y?
`x < y`	Is x less than y?
`x >= y`	Is x greater than or equal to y?
`x <= y`	Is x less than or equal to y?
`x == y`	Is x equal to y?
`x != y`	Is x not equal to y?

Do not confuse = and ==

- The comparisson for equality is two = symbols together: **==**
- The assignment operator is one = symbol: **=**
- Examples:

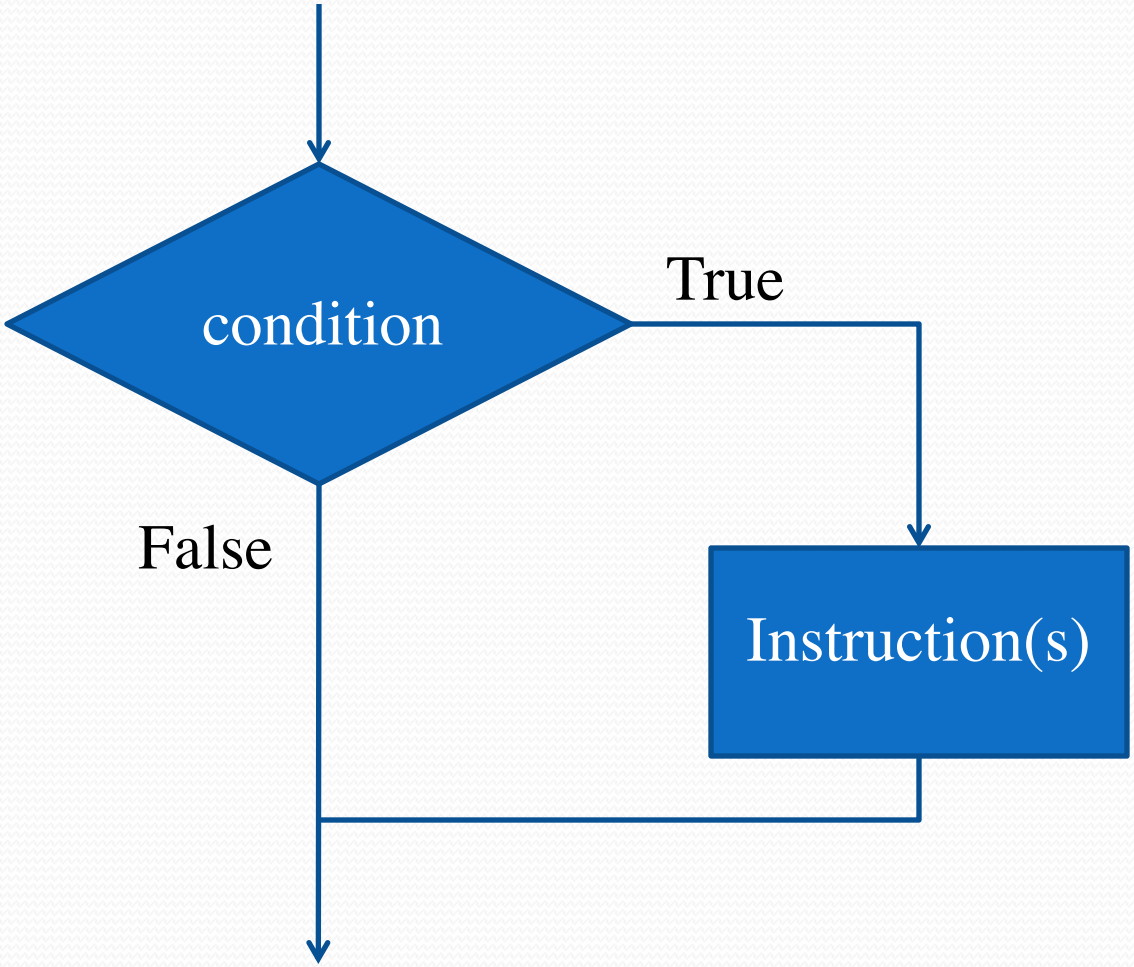
```
x = 25  #assignment, x holds the value 25
x == 25 #comparison, is x equal to 25?
```

- An assignment is **True** if it is successful.
- Mixing up **=** and **==** is a very common error.
  - Using = instead of == creates logical errors

The `if` statement

- The `if` statement is used to create a decision structure
  - which allows a program to have more than one path of execution (branches).
- The `if` statement causes one or more statements to execute only when a Boolean expression is true
- In Python the statements in a branch must be indented some number of spaces
  - Typically four spaces
  - IDEs will automatically indent for you

Simple `if` statement



How does the `if` statement work?

- First the condition is evaluated
- If the result of the condition is `True` the block of instructions is performed
- If the result of the condition is `False` the block of instructions is skipped (ignored)

- Conclusion: the block of instructions associated to the `if` is performed only when the result of the condition is true

Simple `if` statement

- Syntax:

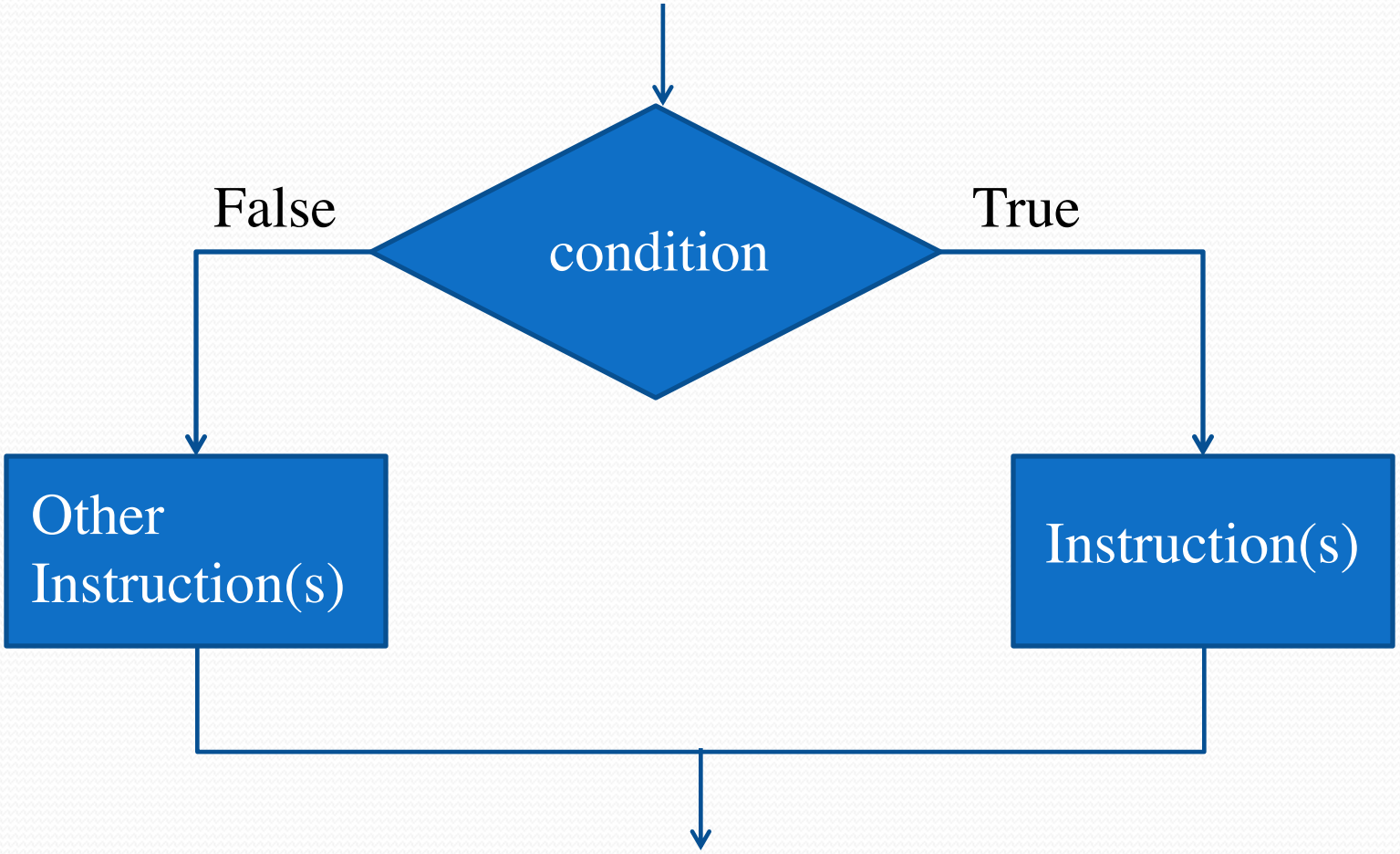
```
if condition :
  Instruction(s)
```

- Examples:

```
if (hours > 40) :
  extraHours = hours - 40
  overtime = extraHours * 1.5 * hourlyRate

if grade < 60 :
  print ("Failed")
```

The `if-else` statement



How does if-else statement work?

- First the condition is evaluated
- If the result of the condition is `True` one block of instructions is executed.
- If the result of the condition is `False` the other block of instructions executed.

The `if-else` statement

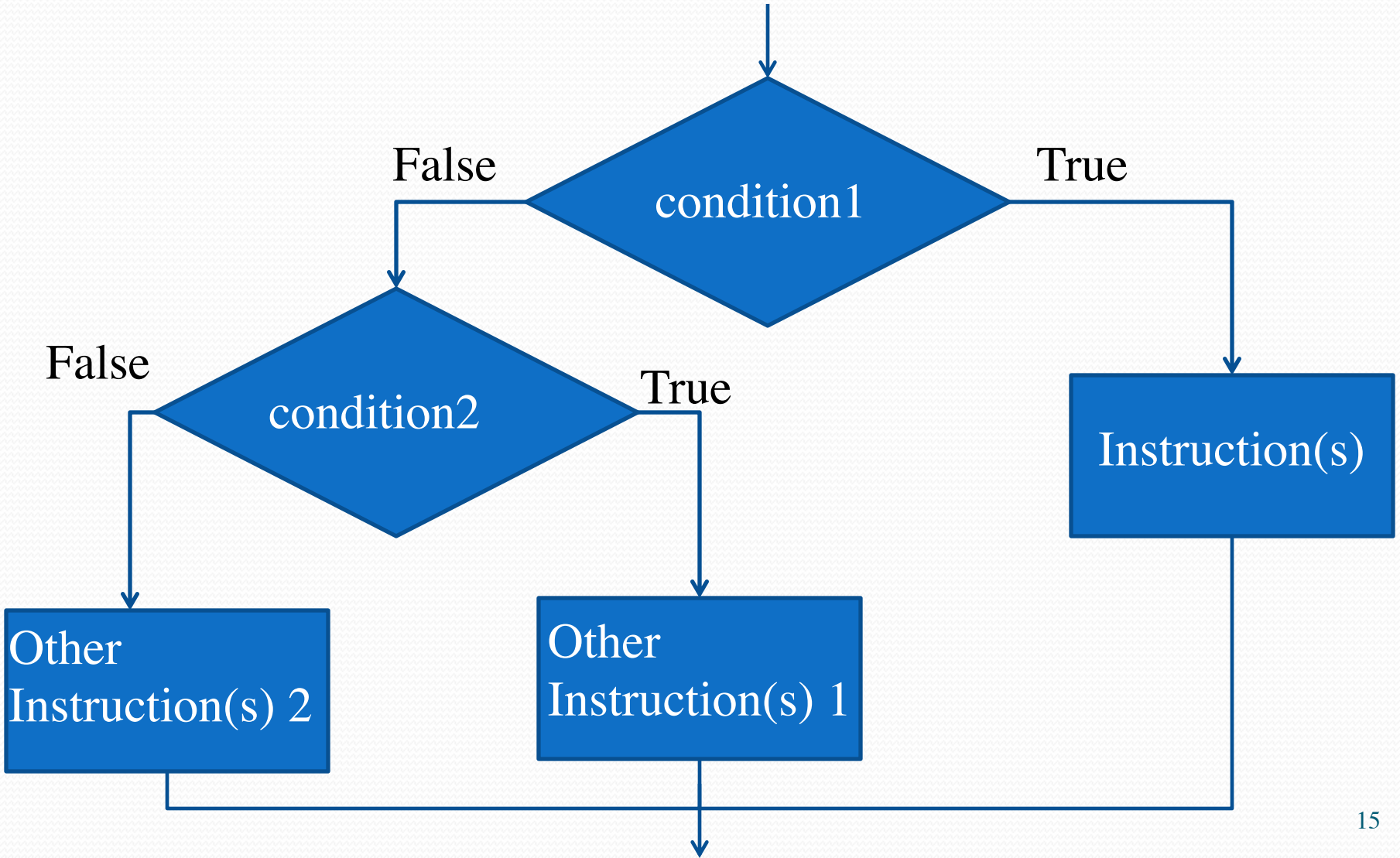
- Syntax:

```
if condition :
  Instruction(s)
else :
  Other instruction(s)
```

- Example:

```
number = int(input('Enter a number: '))
if (number % 2 == 0) :
  print(number, 'is even')
else :
  print(number, 'is odd')
```

Multi-branch `if-else` statements



How do multi-branch statements work?

- We can extend the `if-else` statement to have as many branches as needed.
- `elif` is used to add additional conditions.
  - `elif` is short for else if.
- Each branch's condition is checked in sequence.
- As soon as one condition is `True`, the branch's statements are executed
  - No subsequent branch is considered
- If none of the conditions is `True`, the `else` branch executes.

Multi-branch `if-else` statements

- Syntax:

```
if condition1 :
  Instruction(s)
elif condition2 :
  Other instruction(s)1
else :
  Other instruction(s)2
```

- Example:

```
if (number > 0) :
   print('Positive')
elif (number < 0) :
   print('negative')
else :
    print('Zero')
```

Comparing Strings

- Strings can be compared using the `==` and `!=` operators
- Strings can also be compared using `>`, `<`, `>=`, `<=`
- Strings are compared character by character based on the ASCII/Unicode values for each character
  - Lexicographic order
  - If the shorter word is substring of a longer word, the longer word is greater than the shorter word
- String comparisons are case sensitive

Comparing Strings: Examples

```
s = "Ana"
r = "Amy"
if s == r :
  print("Same name")

if s != r :
  print("Different names")

print("In alphabetical order")
if s > r : #Ana > Amy
  print(r, s)
else :
  print(s, r)
```

In ASCII/Unicode:

- Ana: 65 110 97
- Amy: 65 109 121
- Amy is less than Ana
  Ana is greater than Amy

Comparing Strings

- The **`lower()`** and **`upper()`** string methods can change the case of a string to upper or lower.
- Example:

```
>>> answer = input('Continue (yes/no)? ')
Continue (yes/no)? YeS
>>> answer == 'yes'
False
>>> answer.lower() == 'yes'
True
```

Floating-Point Comparison

- Floating-point numbers should not be compared using ==
- Why? Some floating-point numbers cannot be exactly represented in the limited available memory bits.
  - Floating-point numbers expected to be equal may be close but not exactly equal.
- Example:

```
>>> value = 10 / 3
>>> print(value)
3.3333333333333335
>>> value == 3.3333
False
>>> value == 3.33333333
False
```

Floating-Point Comparison

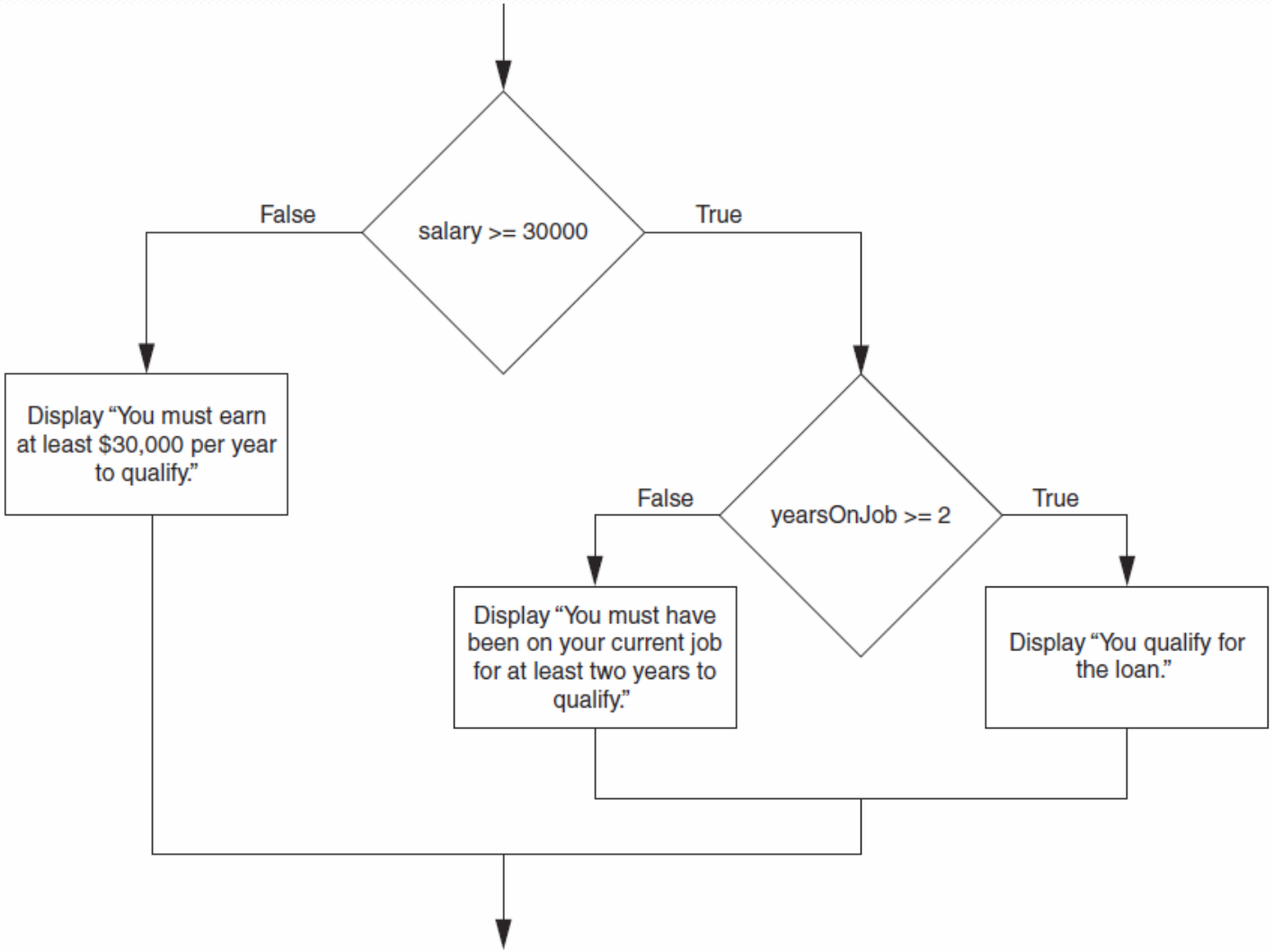
- Floating-point numbers should be compared for "close enough" rather than exact equality.
- The difference threshold indicating that floating-point numbers are equal is often called the epsilon.
- Epsilon's value depends on the program's expected values, but 0.0001 is common.
- Example:

```
>>> value = 10 / 3
>>> epsilon = 0.00005
>>> (abs(value - 3.333333)) < epsilon
True
```

Nested Decision Structures

- A decision structure can be nested inside another decision structure
  - Commonly needed in programs
  - Used to refine the decision process
- Example:
  - Determine if someone qualifies for a loan, they must meet two conditions:
    - Must earn at least $30,000 a year
    - Must have been employed for at least two years
  - Check first condition, and if it is true, check second condition

Nested Decision Structures



Example

```
#determine whether the customer qualifies for a loan
minSalary = 30000
minYears = 2
salary = int(input('Enter current salary: '))
yearsOnJob = int(input('Enter number of years at your job: '))

if salary >= minSalary :
    if yearsOnJob >= minYears :
        print('You qualify for the loan')
    else :
        print('You must have been on your current job for at
              least', minYears, 'to qualify')
else :
    print('You must earn at least', minSalary, 'to qualify')
```

Catching input errors

- A program should not crash because the user provides erroneous input to our program.
- If statements can be used to perform basic input validation
  - Check if the user entered the expected input
- Example:

```
number = int(input("Enter a value greater than zero: "))
if number > 0 :  #good input
   print(math.sqrt(number))
else :  #bad input – send error message
   print("Invalid input! Cannot perform calculation")
```

Catching input errors

- Sometimes the problem is that the user enters the wrong kind of data:

```
>>> weight = int(input('Enter your weight: '))
Enter your weight: one hundred and twenty
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'one
hundred and twenty‘
```

- To handle this type of errors (exceptions) and avoid our programs crashing, we use the **`try`** and **`except`** construct

Try / Except

```
try:
    # Normal code that might produce errors
except Exception as e: #Go here if error occurs in try block
    # Exception handling code
```

- When a **try** is reached, the statements in the try block are executed.
- If no exception occurs, the **except** block is skipped and the program continues.
- If an exception does occur, the **except** block is executed, and the program continues after the **except** block.
- Any statements in the **try** block not executed before the exception occurred are skipped.

Example

```
try:
    number = int(input("Enter a value greater than zero: "))
    if number > 0 :
       print(math.sqrt(number))
    else :
      print("Invalid input! Cannot perform calculation")

except Exception as e:
    print('could not read number')
    print('Error:', e)
```

Common Error Types

Error Name	Description
`Exception`	General for all non-critical errors
`ZeroDivisionError`	Division by zero attempted
`NameError`	A variable name was used that doesn't exist
`ValueError`	A type conversion failed

Boolean Logic

- A value that is only `True` or `False` is called a Boolean value.
- Python uses `True` and `False`.
- Notes:
  - These are keywords
  - The first letter must be capital.

Boolean Operators

- There are three Boolean operators.
  - `not`
  - `and`
  - `or`
- These can be used to make complex conditions.

`not` operator

- Reverses the value of truth

P	not P
True	False
False	True

- Example: for `x is not greater than 5` write:

```
not (x > 5)
```

`and` operator

- **`True`** only when both parts are **`True`**

P	Q	P and Q
True	**True**	**True**
True	False	False
False	True	False
False	False	False

- Example: for `x greater than 10 and y less than 8` write

```
(x > 10) and (y < 8)
```

`or` operator

- **`True`** when either part is **`True`**

P	Q	P or Q
True	True	True
True	False	True
False	True	True
False	**False**	**False**

- Example: for `x greater than 20 or x less than 10` write

```
(x > 20) or (x < 10)
```

More Examples

```
>>> (temperature <= 32 and humidity > 80)
>>> (choice == 'A' or choice == 'a')
>>> (grade >= 90 or gpa >= 3.50 )
>>> (not (choice == 'A' or choice == 'a'))
>>> isPositive = number >= 0 #Boolean variable
>>> continue = False
```

Precedence Rules

Operator/Convention	Description
( )	Items within parentheses are evaluated first
* / % + -	Arithmetic operators (using their precedence rules; see earlier section)
< <= > >= == !=	Relational, (in)equality, and membership operators
not	not (logical NOT)
and	Logical AND
or	Logical OR

Ternary operation

- Also known as conditional expression
- Takes three operands (ternary)
- Yields a value
- Format:

```
ValueToYieldWhenTrue if condition else ValueToYieldWhenFalse
```

- Example:

```
result = (n - 1) if (n > 5) else (n + 1)
absolute = ( x ) if ( x >= 0 ) else ( -x )
```

Membership Operators

- You can use the `in` operator to determine whether an item is contained in a container
  - String, list, tuple, dictionary, set
- Syntax: *`item in container`*
  - This expression returns `True` if item is found in the container, `False` otherwise
- You can use the `not in` operator to determine whether an item is contained in a container
  - String, list, tuple, dictionary
- Syntax: *`item not in container`*
  - This expression returns `True` if item is not found in the sequence, `False` otherwise

Membership Examples

```
fruits = ["banana", "cherry", "pear", "apple"]
search = "pear"
if search in fruits :
      print (search, " was found in fruits")
else :
      print (search, " was not found in fruits")

letter = “u”
if letter in “aeiouAEIOU”
      print (letter, “is a vowel”)
letter2 = ‘y’
if letter2 not in “aeiouAEIOU”
      print (letter2, “is not a vowel”)
```

Membership Operators

Note: In a dictionary the membership operators check for a specific key, not for the values associated with the key

```
states = {'Iowa' : 'IO', 'Ohio' : 'OH', 'Georgia' : 'GA'}
if 'Georgia' in states :
   print("Georgia's abbreviation is", states['Georgia’])
else :
 print(‘This state is not in the dictionary’)
```

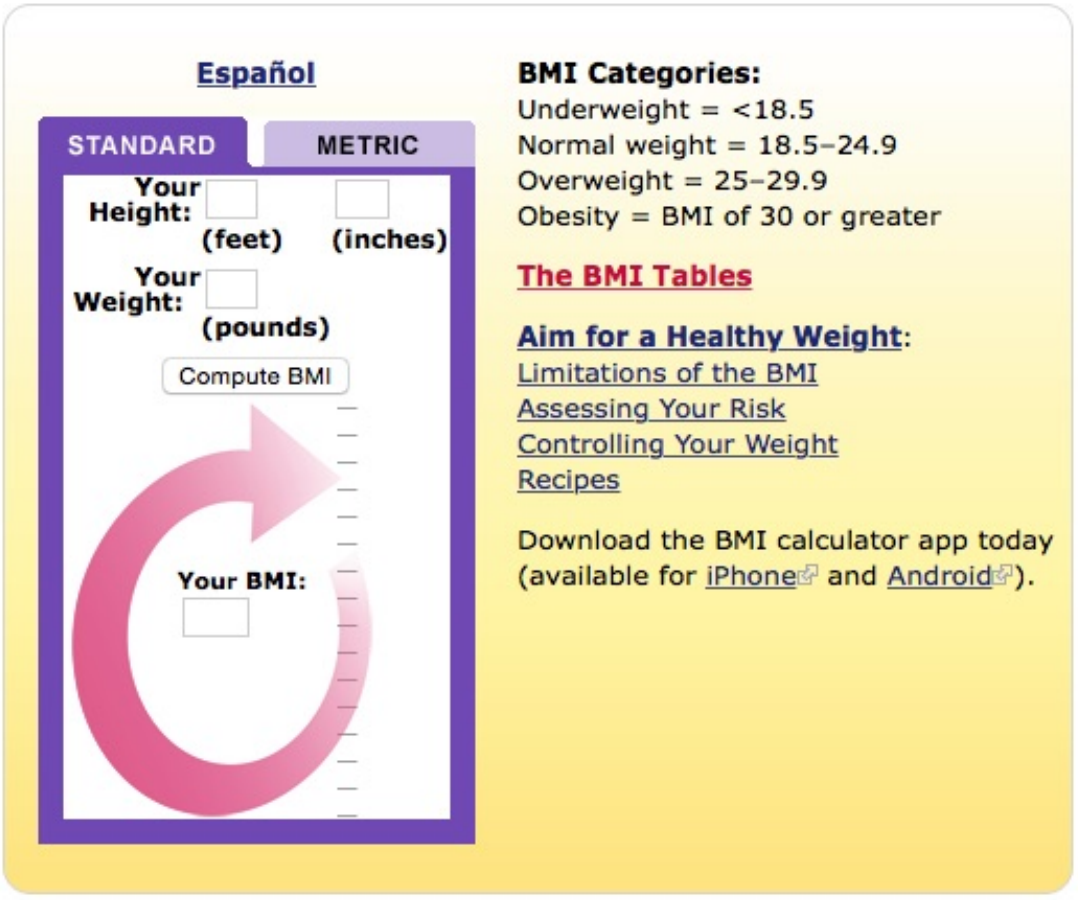
- If you need to find out what are the keys in a dictionary, you can use the keys() command:

```
>>> print(states.keys())
dict_keys(['Iowa', 'Ohio', 'Georgia'])
```

Example: BMI Calculator

- Body Mass Index: BMI
- An easy way to gauge if a person is overweight
- BMI has well known flaws and advantages:
  - Advantage: Easy and Quick to Determine
  - Disadvantage: Does not account for details of health/body type

Example: BMI Calculator



https://www.nhlbi.nih.gov/health/educational/lose_wt/BMI/bmicalc.htm

BMI Formula

- The basic formula is straightforward.

$$\text{BMI} = \frac{Weight\ in\ Kg}{(Height\ in\ meters)^2}$$

http://extoxnet.orst.edu/faqs/dietcancer/web2/twohowto.html

BMI script initial plan

- Input: the user probably knows their height/weight in feet/pounds
- Convert Units to Kg and meters
- Do Calculations: using formula provided
- Determine status of user (normal, overweight, underweight, obese): using the BMI table

Possible issues

- Weight is not a number
- Weight is a negative number
- Height is not a number
- Height is a negative number

BMI script refined plan

1. Initial Setup
2. Ask for Weight (pounds)
3. Check weight for errors
4. Ask for Height (feet)
5. Check height for errors
6. Exit if any errors happened
7. Convert Units to Kg and meters
8. Determine BMI (using formula)
9. Determine weight status (using BMI table)

See example: `bmiCalculator.py`

Possible Revisions

- Allow repeated mistakes (Covered Next Week)
- Allow the user to enter feet and inches for height
- Ask for another person's values (Next Week)