CSE/ISE 337
Scripting Languages

Python 02:

Control Structures: if, elif, else; while; for

Function Definitions

Previous Lecture

- Working with the Python Shell
- Running Python programs from the command-line
  - $ python<ver-name> /path/to/program.py
- Python basics
  - Datatypes
  - Variables
  - function calls
  - Assignment
  - Modules
  - packages

Conditional Statement

- Conditional statements are used to express conditional computation
  - *if* statement
  - *else* statement
  - *elif* statement
  - Nested conditionals

if Statement

- Syntax:

```
if <boolean-expression>:
    <block-of-statements>
```

- Example:

```
if n >= 95:
    print('A')
```

else Statement

- Syntax:

```
if <boolean-expression>:
  <block-of-statements>
else:
  <block-of-statements>
```

- Example:

```
if n >= 95:
  print('A')
else:
  print('A-')
```

elif Statement

- Syntax:

```
if <boolean-expression>:
  <block-of-statements>
elif <boolean-expression>:
  <block-of-statements>
elif …
else:
  <block-of-statements>
```

- Example:

```
if n >= 90:
  print('A')
elif n >= 80:
  print('B')
elif n >= 70:
  print('C')
else:
  print('D')
```

Nested Conditionals

- Syntax

```
if <boolean-expression>:
  <block>
  if <boolean-expression>:
    <block>
  …
```

Who does the *else* belong to?

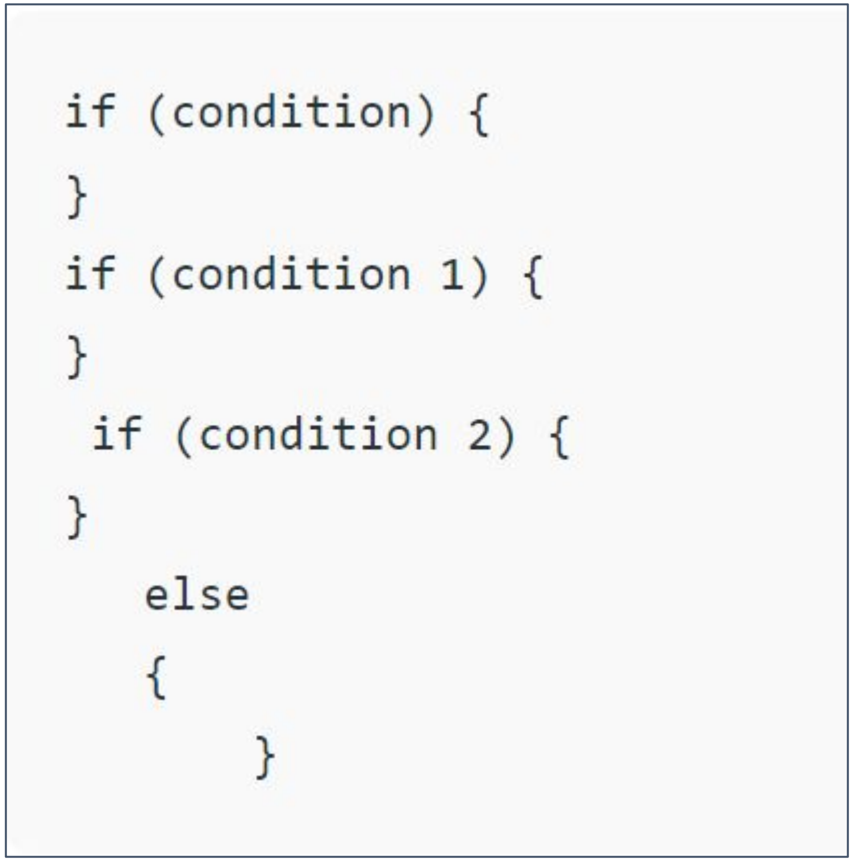
- Example:

```
if n >= 90:
  if n >= 95:
    print('A+')
  else:
    print('B')
```

Dangling else problem

- Syntactic ambiguity
- The dangling else problem results when a programmer believes (or forgets) the code indentation and loses track of the if statement that goes with the else statement.
- In the example, there are multiple “*ifs”* with multiple conditions and here we want to pair the outermost if with the else part. But the else part doesn’t get a clear view with which “*if”* condition it should pair. This leads to inappropriate results in programming.

Example



```
if (condition) {
}
if (condition 1) {
}
 if (condition 2) {
}
   else
   {
       }
```

Indentation

- Indentation is Python's way of grouping statements.
- Multiple statements of the same indentation belong to the same group.
- Indent via consistent white space
  - Spaces or tabs (DON'T MIX THEM UP!)
- Incorrect indentation leads to syntax or logical errors.

```
if n >= 90:
  if n >= 95:
    print('A+')
  if n < 95:
    print('A')
print('End')
```

Common Operators Used in Conditions

- Logical operators: *and, or, not*
- Relational operators: `<, <=, ==, !=, >=, >`

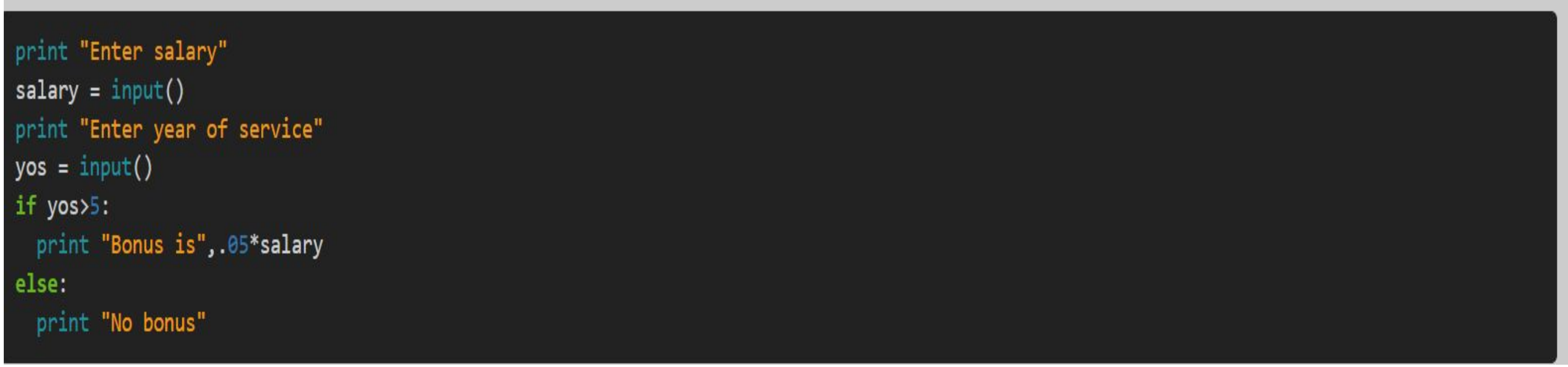
```
if (n >= 90 and n < 95):
if (n >= 90 or n < 95):
if not (n > 90 and n == 90):
```

Example Script

- See `grade.py`

Example Script

- A company decided to give bonus of 5% to employee if his/her year of service is more than 5 years. Ask user for their salary and year of service and print the net bonus amount.
-



```
print "Enter salary"
salary = input()
print "Enter year of service"
yos = input()
if yos>5:
  print "Bonus is",.05*salary
else:
  print "No bonus"
```

Example Script

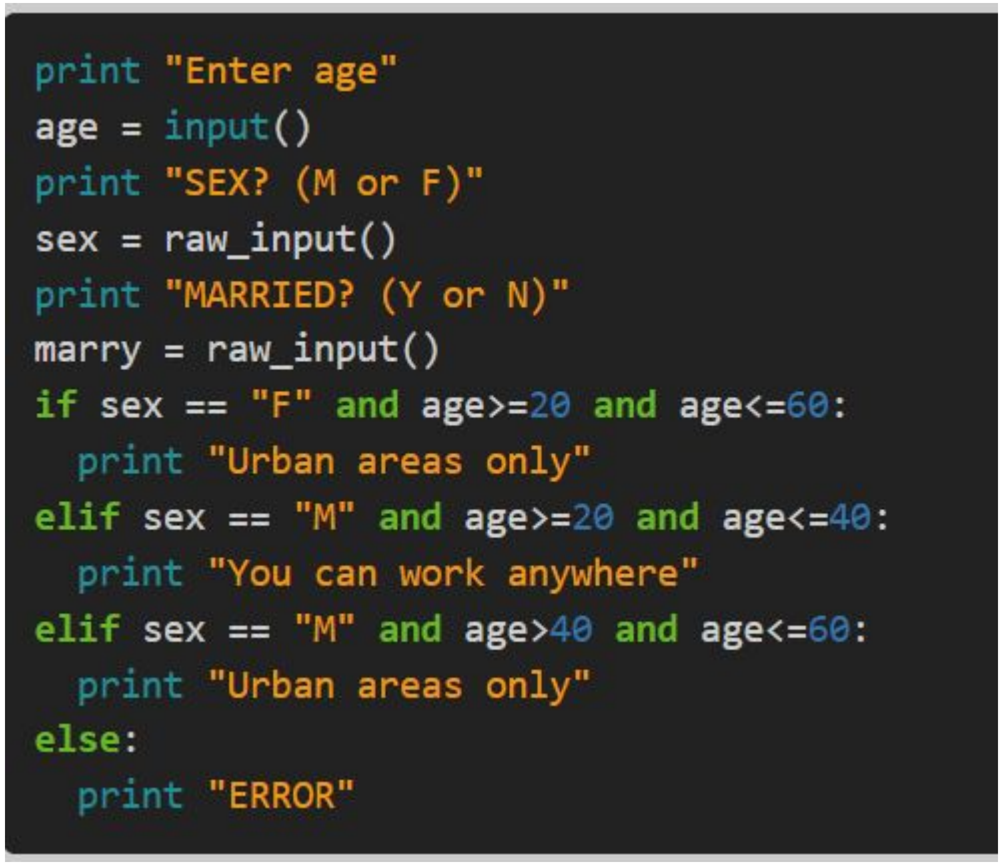
Ask user to enter age, sex (M or F), marital status (Y or N) and then using following rules print their place of service.

if employee is female, then she will work only in urban areas.

if employee is a male and age is in between 20 to 40 then he may work in anywhere

if employee is male and age is in between 40 t0 60 then he will work in urban areas only.

And any other input of age should print "ERROR".



```
print "Enter age"
age = input()
print "SEX? (M or F)"
sex = raw_input()
print "MARRIED? (Y or N)"
marry = raw_input()
if sex == "F" and age>=20 and age<=60:
  print "Urban areas only"
elif sex == "M" and age>=20 and age<=40:
  print "You can work anywhere"
elif sex == "M" and age>40 and age<=60:
  print "Urban areas only"
else:
  print "ERROR"
```

Example Script

Write a program to check if a year is leap year or not.

If a year is divisible by 4 then it is leap year but if the year is century year like 2000, 1900, 2100 then it must be divisible by 400/

Hint→ Use % operator

Question on Conditional statements!

Iteration Constructs

- Loops are used to express repeated computations
    - *while* loops
    - *for* loops

- Python also has recursion, which we will return to when discussing function definitions.

while Statement

- Syntax:

```
while <Boolean-expression>:
  <block>
```

- Example: (What happens if the user enters -1 at the first input?)

```
sum = 0.0
count = 0
num = int(input("Enter a score: "))
while num != -1:
  sum = sum + num
  count = count + 1
  num = int(input("Enter a score: "))
print("The average is: ", sum / count)
```

while Statement

- Syntax:

```
while <Boolean-expression>:
  <block>
```

- Example: (What happens if the user enters -1 at the first input?)

```
sum = 0.0
count = 0
num = int(input("Enter a score: "))
while num != -1:
  sum = sum + num
  count = count + 1
  num = int(input("Enter a score: "))
print("The average is: ", sum / count)
```

for Statement

- Syntax:

```
for <expression-list> in <collection>:
  block
```

- Example:

```
for i in range(4):
  print(i)
```

Example Script

.Slide19.py

range() Function

- A built-in function that constructs a sequence of integers
- *range(stop)*
  - a sequence of integers from 0 to *stop* – 1
- *range(start, stop)*
  - a sequence of integers from *start* to *stop* - 1
- *range(start, stop, step)*
  - a sequence integers: *start*, *start* + *step*, *start* + (2 * *step*), …, < *stop*
- What happens when we print a range?

Looping Over Strings

- Combine *range()* and *len()*

```
a = 'lectures'
for i in range(len(a)):
  print(a[i])
```

- Or directly

```
a = 'lectures'
for c in a:
  print(c)
```

Example Script

- Slide23.py

Looping Over Strings

- Combine *range()* and *len()*

```
a = 'lectures'
for i in range(len(a)):
  print(a[i])
```

- Or directly

```
a = 'lectures'
for c in a:
  print(c)
```

How would you reverse a string using a loop?

Example script

- reverse25.py

Questions!

https://forms.gle/wcKVHbpVApHFdrBTA

break and *continue* Statements

- *break* terminates a loop

```
for n in range(1, 11):
    for k in range(2, n):
        if n % k == 0:
            print(n, ':', True)
            break
    print(n, ':', False)
```

- *continue* immediate returns to the top of a loop

```
for k in range(1, 11):
    if n % 2 == 0:
        print(n, ':',
True)
        continue
    print(n, ':', False)
```

Example Script

- Slide28.py → break statement
- Slide28_A.py → continuous statement

else Clause in Loops

- The *else* clause in a loop is executed when the loop terminates "naturally".
  - No break statement or other interruption is executed.

```
for n in range(1, 11):
  for k in range(2, n):
    if n % k == 0:
      break
  else:
    print(n, 'is prime.")
```

pass Statement

- The *pass* statement is a no-op (frequently, 'NOP')
  - It literally does nothing.
- Useful in contexts where the program is not required to do anything.
  - Cases that get skipped
  - Unimplemented behavior
- Example:

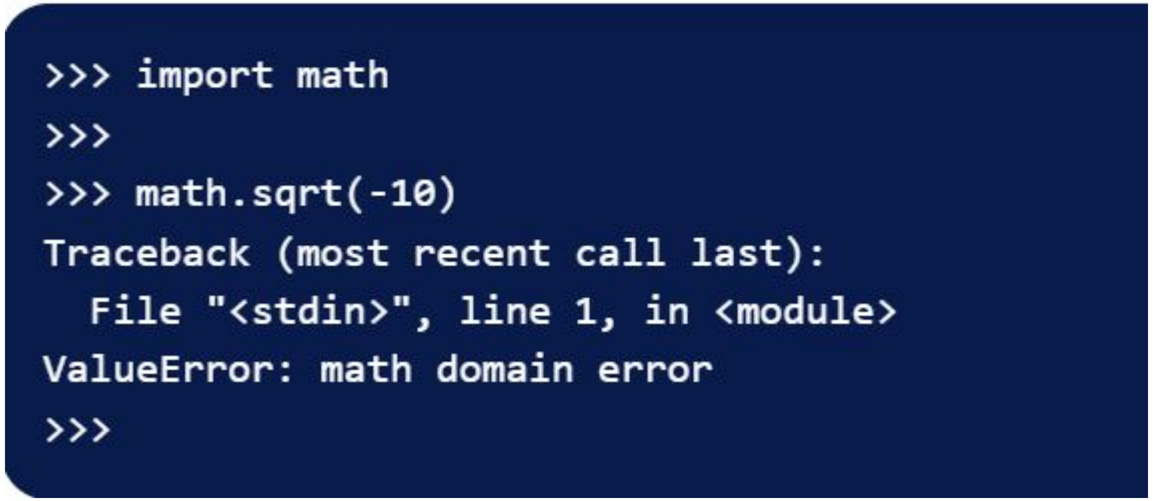
```
for i in range(4):
  pass # implement later
```

Exceptions

- Exceptions are syntactically-correct statements that lead to runtime errors.
- An exception halts program execution.
  - Unless properly handled within the program.
- Python features numerous built-in Exception classes
  - *ZeroDivisionError* – raised when a divide by zero occurs
  - *NameError* – raised when a local or global name is not found
  - *TypeError* – raised when an operation is applied to an object of an inappropriate type
  - *ValueError* – raised when an operation receives an inappropriate value
- https://docs.python.org/3/library/exceptions.html

Example Script

- Slide18.py
- ValueError Exception



```
>>> import math
>>>
>>> math.sqrt(-10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>>
```

Error vs. Exception

An error is an issue in a program that prevents the program from completing its task. In comparison, an exception is a condition that interrupts the normal flow of the program. Both errors and exceptions are a type of runtime error, which means they occur during the execution of a program.

Exception Handling

- Exceptions may be handled to continue program execution.
    - Why?
- Exceptions are handled via *try-except* statements.
    - Very similar to *try-catch* statements in Java
- Syntax:

```
try:
    <block>
except <exception-list>:
    <block>
```

Interpreting *try*-*except* Statements

- First, execute statements in the *try* block
  - If an exception occurs in the *try* block, then skip to the matching *except* block.
  - If no exception occurs in the *try* block, then skip all *except* blocks.
- Example:

```
while True:
  try:
    x = int(input("Enter a number: "))
    break
  except ValueError:
    print("Input must be a number")
```

Multiple Exceptions

- Multiple Exceptions may be raised in a *try* block.
- A sequence of *except* blocks can be defined, each having a distinct exception list.
- If an exception is raised in the *try* block, skip to the first (lexically) matching *except* block.

```
while True:
  try:
    n = int(input('n: '))
    d = int(input('d: '))
    print('Q =', n // d)
    break
  except (ValueError, TypeError):
    print('Only numbers allowed')
  except ZeroDivisionError:
    print('d cannot be 0')
```

Catch-all Exception

- If an exception type is not specified for an *except* clause, it will handle any runtime exception.

```
while True:
  try:
    n = float(input('n: '))
    d = float(input('d: '))
    print('Result =', n / d)
    break
  except:
    print('Bad Input')
```

Example Script

- Slide38.py

Defining Functions

- A function definition begins with the keyword *def*, followed by a name (identifier), a parenthesized list of formal parameters, followed by a ':'
- An indented block of statements makes up the body of the function.
- Formal parameters must have names and may be assigned a default value.
- A function always returns a value
  - *return <expression>* - returns the value of *<expression>*
  - *return* – returns *None*
  - If no return statement is reached, the function returns None when it terminates.

Function Definition Examples

```
def fib(n):
  a, b = 0, 1
  while a < n:
    print(a, end = ', ')
    a, b = b, a + b
```

```
def fib(n):
  a, b, result = 0, 1, ''
  while a < n:
    result = result + a + ', '
    a, b = b, a + b
  return result
```

https://en.wikipedia.org/wiki/Fibonacci_sequence

Another Function Definition

- Default values for formal parameters

```
def fib(n, start = 1):
  a, b = 0, start
  while a < n:
    print(a, end = ' ')
    a, b = b, a + b
```

docstring Statement

- Documentation strings or *docstrings* are used to add documentation to function definitions.
- Documentation strings make code more readable.
- *docstrings* are enclosed within triple quotes as the first line in the body of the function.
- Characters inside the *docstring* are not executed by the interpreter, so they are sometimes used as multi-line comments in Python.

docstring Example

```
def fib(n):
  ''' input: An integer n
      output: Fibonacci sequence up to n
  '''
  a, b, result = 0, 1, ''
  while a < n:
    result = result + a + ' '
    a, b = b, a + b
  return result
```

Calling Functions

- A function is called by specifying its name and arguments

- Arguments may be identified in two ways:
  - By Position
  - By Keyword

- Default arguments may be skipped.
  - If they do not prevent identifying other arguments by position.

Function Call Examples

- `def fib(n, start = 1):`
  - `fib(100)`
  - `fib(100, 24)`
  - `fib(n = 100, start = 24)`
  - `fib(start = 24)`
  - `fib(start = 24, n = 100)`
  - `fib(100, start = 24)`
  - `fib(start = 24, 100)`

- Which of these work? Which do not?

Next time

- Quiz
- Python Function and Data Structure