Lecture 2

Welcome to ECL 205

Object Oriented Programming

Asst. Prof. Snigdha Bhagat

PREVIOUS LECTURE

- syntax and semantics
- scalar objects
- simple operations
- expressions, variables and values

TODAY

- string object type
- branching and conditionals
- indentation
- iteration and loops

STRINGS

- letters, special characters, spaces, digits
- enclose in **quotation marks or single quotes**

```
hi = "hello there"
```

- **concatenate** strings

```
name = "Burkay"
greet = hi + name
greet
```

```
## 'hello thereBurkay'
```

```
greeting = hi + " " + name
greeting
```

```
## 'hello there Burkay'
```

- do some **operations** on a string as defined in Python docs

```
silly = hi + " " + name * 3
silly
```

```
## 'hello there BurkayBurkayBurkay'
```

STRING LENGTH

- the length of a string can be computed by `len()`

```
name = "Burkay"
len(name)
```

```
## 6
```

```
surname = "Genc"
fullname = name + " " + surname
len(fullname)
```

```
## 11
```

STRING INDEXING

- You can get a specific character in a string by indexing the string
- An **index** is a position in a string
- **In Python indices start from 0**
  - The first character is at index 0
  - The second character is at index 1
  - ...

```
fullname
```

```
## 'Burkay Genc'
```

```
fullname[0]
```

```
## 'B'
```

```
fullname[1]
```

```
## 'u'
```

```
fullname[5]
```

```
## 'y'
```

EXERCISE

- How do you print the last character of a string?

EXERCISE

- If there are **k** characters in a string
  - the last character is at index **k-1**

```
lastCharacterIndex = len(fullname) - 1
fullname[lastCharacterIndex]
```

```
## 'c'
```

- Python uses **negative indices** to **count from the end**
  - So, a simpler solution is :

```
fullname[-1]
```

```
## 'c'
```

SLICING

- You can get a subset of characters by slicing
- To slice a string you provide
  - the **starting index** of the slice (inclusive)
  - separate with a column, **:**
  - the **ending index** of the slice (exclusive)

```
# The string is   : Burkay
# The indices are : 012345

name[2:4]
```

```
## 'rk'
```

- So, `str[i:j]` gives you characters at indices:
  - `i, i+1, i+2, ..., j-1`

SLICING

- You can omit the stating or the ending indices, or both

```
fullname
```

```
## 'Burkay Genc'
```

```
fullname[:5]
```

```
## 'Burka'
```

```
fullname[5:]
```

```
## 'y Genc'
```

```
fullname[:]
```

```
## 'Burkay Genc'
```

SLICING

- You can even slice with negative indices:

```
fullname[-4:]
```

```
## 'Genc'
```

```
fullname[:-4]
```

```
## 'Burkay '
```

```
fullname[2:-2]
```

```
## 'rkay Ge'
```

INPUT/OUTPUT: print

- used to **output** stuff to console
- keyword is `print`

```
x = 1
print(x)
```

```
## 1
```

- `print` adds a space between parameters. To avoid the space, you should concatenate the parameters yourself.

```
print("my fav num is", x, ".", "x =", x)
```

```
## my fav num is 1 . x = 1
```

```
x_str = str(x)
print("my fav num is " + x_str + ". " + "x = " + x_str)
```

```
## my fav num is 1. x = 1
```

INPUT/OUTPUT: input("")

- prints whatever is in the quotes
- user types in something and hits enter
- binds that value to a variable

```
text = input("Type anything... ")
print(5*text)
```

```
## Type anything... 5
## 55555
```

- `input` **gives you a string** so you must cast if working with numbers

```
num = int(input("Type a number... "))
print(5*num)
```

```
## Type a number... 5
## 25
```

EXERCISE

- Write a program that will ask your_name and your_age and then print out

`your_name is your_age years old.`

- For example:

```
## What is your name? Burkay
## How old are you? 41

## Burkay is 41 years old.
```

BRANCHING

COMPARISON OPERATORS

- `i` and `j` are variable names
- comparisons below evaluate to a Boolean (true/false)

```
i > j
i >= j
i < j
i <= j
```

- **equality test**, `True` if `i` is the same as `j`

```
i == j
```

- inequality test, `True` if `i` is **not** the same as `j`

```
i != j
```

COMPARISON OPERATORS

```
5 > 3
```

```
## True
```

```
a = 7
b = 5
b >= a
```

```
## False
```

```
a == b
```

```
## False
```

```
a == b + 2
```

```
## True
```

- Python is first computing expressions, then checking comparisons

```
a != b
```

```
## True
```

COMPARISON OPERATORS

```
b = "burkay"
c = "cemal"
d = "davut"
b == c
```

```
## False
```

```
c < d
```

```
## True
```

```
b > d
```

```
## False
```

```
b = "burkay"
B = "Burkay"
B < b
```

```
## True
```

```
b < B
```

```
## False
```

- Capital letters come first

LOGIC OPERATORS

- `a` and `b` are variable names (with Boolean values)
- `not` negates the boolean value
- `and` returns `True` only if both operands are `True`
- `or` returns `False` only if both operands are `False`

a	b	not a	a and b	a or b
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

LOGIC OPERATORS

```
a = 5
b = 7
c = 3
a > b
```

```
## False
```

```
not (a > b)
```

```
## True
```

```
(a > b) and (b > a)
```

```
## False
```

```
(a > b) or (b > a)
```

```
## True
```

```
(a < c) or ((b > c) and not (a < c))
```

```
## True
```

COMPARISON EXAMPLE

```
pset_time = 15
sleep_time = 8
print(sleep_time > pset_time)
```

```
## False
```

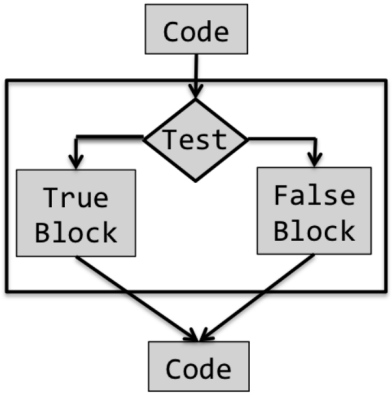
```
derive = True
drink = False
both = drink and derive
print(both)
```

```
## False
```

- *derive* obtain (a function or equation) from another by a sequence of logical steps, for example by differentiation.

BRANCHING PROGRAMS

BRANCHING



We can define **branches** in the flow of a program using **conditions**.

- We use the `if/else` keywords for that:

```
if <condition>:
   <expression>
   <expression>
   ...
elif <condition>:
   <expression>
   <expression>
   ...
else:
   <expression>
   <expression>
   ...
```

BRANCHING

- We can also create multiple branches using `elif` keyword:

```
if <condition>:
   <expression>
   <expression>
   ...
elif <condition>:
   <expression>
   <expression>
   ...
elif <condition>:
   <expression>
   <expression>
   ...
else:
   <expression>
   <expression>
   ...
```

BRANCHING EXAMPLE

```
a = 5
b = 3
if (a < b):
   print("a is less than b.")
else:
   print("a is greater than b.")
```

```
## a is greater than b.
```

BRANCHING EXAMPLE

- Returning the maximum of two numbers

```
a = 7
b = 3

if a > b:
   print(a)
else:
   print(b)
```

```
## 7
```

- What if they are equal?

```
a = 7
b = 3

if a > b:
   print(a)
elif a < b:
   print(b)
else:
   print("they are equal")
```

```
## 7
```

INDENTATION

- matters in Python
- how you denote blocks of code

```
# block level 1
do_something()
if True:
   # block level 2
   do_something()
   if True:
      # block level 3
      do_something()
   # block level 2
   do_something()
# block level 1
do_something()
```

INDENTATION

```
if False:
   print("This shouldn't print.")
print("But this should.")
```

```
## But this should.
```

vs.

```
if False:
   print("This shouldn't print.")
   print("And this shouldn't print as well.")
```

INDENTATION

- We can **nest** if statements to create complex branching
- Nesting is done via indentation

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
   print("x and y are equal")
   if y != 0:
      print("therefore, x / y is", x/y)
elif x < y:
   print("x is smaller")
else:
   print("y is smaller")
print("thanks!")
```

- Exercise
  - Execute this code in Python and test with different inputs.
  - Find a pair of values for x and y for each possible output.

WARNING: ASSIGNMENT vs. EQUALITY

- Be careful not to confuse = and ==
- = is assignment
  - righthand side value is assigned to lefthand side variable
- == is a check for equality
  - returns True if lefthand side is equal to righthand side

```
x = 5
if x == 5:
   print("This will be accepted and printed.")
```

```
## This will be accepted and printed.
```

```
if x = 5:
   print("This won't be accepted by Python.")
```

```
## Error: invalid syntax (<string>, line 1)
```

ITERATION

A THOUGHT EXERCISE

- Ask the user for a number and then print that many X's

```
numberOfXs = int(input("Enter a number:"))

if numberOfXs == 1:
   print("X")
elif numberOfXs == 2:
   print("XX")
elif numberOfXs == 3:
   print("XXX")
elif numberOfXs == 4:
   print("XXXX")
# ...
# where to stop???
```

- Can we be smarter than this?
  - Yes!

A THOUGHT EXERCISE

- A better solution is:

```
numberOfXs = int(input("Enter a number:"))
print(numberOfXs * 'X')
```

- We use a property of Python to solve this.
  - We can multiply strings by numbers to repeat them
- But will this always work?
  - No!

ANOTHER THOUGHT EXERCISE

- Ask for a number and **print numbers from 1 to that number**

```
aNumber = int(input("Enter a number:"))

if aNumber == 1:
   print("1")
elif aNumber == 2:
   print("1 2")
elif aNumber == 3:
   print("1 2 3")
elif aNumber == 4:
   print("1 2 3 4")
elif aNumber == 5:
   print("1 2 3 4 5")
## Where to stop???
```

- What to do now?

ITERATION

- When we want a program to **do something many times** following a certain **structure** we use iteration
- Examples
  - `For each student in this class, ask his name`
  - `Water every flower in this room`
  - `For each number from 1 to 10, compute square of that number`
  - `Given k, compute k!`
- All of these require a repetitive task to be done many times
- Python allows repetitive tasks to be easily coded by **looping**

WHILE

- to do something repetitively until we hit a **termination condition** we use the `while` loop

```
while <condition>:
   do_something
```

- Python will `do_something` **again and again** until `<condition>` becomes `False`
  - Beware! If `<condition>` never becomes `False`, then the program **executes forever!**

```
a = 0
while a < 5:
   print(a)
   a = a + 1   # Don't forget this!
```

```
## 0
## 1
## 2
## 3
## 4
```

WHILE

- You can iterate both ways:

```
a = 10
while a > 0:
   print(a)
   a = a - 1
```

```
## 10
## 9
## 8
## 7
## 6
## 5
## 4
## 3
## 2
## 1
```

EXERCISE

- Compute the factorial of a given number

SOLUTION

```
number = 6
factorial = 1  # starting with 1 is important!

while number > 1:
   factorial = factorial * number
   number = number - 1

print(factorial)
```

```
## 720
```

OR

```
number = 6
factorial = 1  # starting with 1 is important!
i = 1

while i <= number:  # why <= and not < ???
   factorial = factorial * i
   i = i + 1

print(factorial)
```

```
## 720
```

EXERCISE

- Find all numbers perfectly dividing a given number

SOLUTION

- Find all numbers perfectly dividing a given number

```
number = 36
i = 1

while i <= number:
   if number % i == 0:
      print(i)
   i = i + 1
```

```
## 1
## 2
## 3
## 4
## 6
## 9
## 12
## 18
## 36
```

EXERCISE

- Print all numbers that are less than 100 and divisible by 5, 7 or 11.

SOLUTION

- Print all the numbers that are less than 100 and divisible by 7, 13 or 19.

```
i = 1
while i < 100:
   if i % 7 == 0:
      print(i, " ", end = "")
   elif i % 13 == 0:
      print(i, " ", end = "")
   elif i % 19 == 0:
      print(i, " ", end = "")
   i = i + 1
```

```
## 7  13  14  19  21  26  28  35  38  39  42  49  52  56  57  63  65  70  76  77  78  84  91
95  98
```

- Don't do the following:

```
i = 1
while i < 100:
   if i % 7 == 0:
      print(i, " ", end = "")
   if i % 13 == 0:
      print(i, " ", end = "")
   if i % 19 == 0:
      print(i, " ", end = "")
   i = i + 1
```

- Why not? How is it different?