
# Intro to Python

## Class 2

 

# Review

- Arithmetic and variables
- Data types
- Text editor, command line, and python shell

# What we will cover today

- Boolean Expressions and Conditionals
- Loops
- Functions

# Boolean Expressions

We can tell the computer to compare values and return True or False. These are called **Boolean expressions**

- Test for equality by using `==`. We can't use `=` because that is used for assignment
- Test for greater than and less than using `>` and `<`

```
a = 5
b = 5
print a == b
# Combine comparison and assignment
c = a == b
print c

print 3 < 5
```

# Boolean Expressions continued

The following chart shows the various Boolean operators

| | |
|---|---|
| a == b | a is equal to b |
| a != b | a does not equal b |
| a < b | a is less than b |
| a > b | a is greater than b |
| a <= b | a is less than or equal to b |
| a >= b | a is greater than or equal to b |

```
a = 3
b = 4
print a != b
print a <= 3
print a >= 4
```

Remember: Equals does not equal "equals equals"

# Conditionals

When we want different code to execute dependending on certain criteria, we use a **conditional**

We achieve this using **if** statements

```
if x == 5:
    print 'x is equal to 5'
```

We often want a different block to execute if the statement is false. This can be accomplished using **else**

```
if x == 5:
    print 'x is equal to 5'
else:
    print 'x is not equal to 5'
```

# Indentation

In Python, **blocks** begin when text is indented and ends when it returns to the previous indentation

Let's look at the previous example again with a few minor changes and examine the meaning of its indentation

```
if x == 5:
    print 'x is equal to 5'
    x_is_5 = True
    print 'Still in the x == 5 block'
else:
    print 'x is not equal to 5'
    x_is_5 = False
    print 'Still in the else block of x == 5'
print 'Outside of the if or else blocks.'
print 'x_is_5:'
print x_is_5
```

# Predicates

The expression after `if` and before the colon is a Boolean expression, also called a **predicate**

A variable can be used to store the evaluation of a predicate, especially if it is particularly long or complicated

Predicates can be combined or prefaced with **Logical operators**: `not`, `and` and `or`

Predicates and the logical operators that act on them can be wrapped in parenthesis to enforce prescendence

# Predicates continued

The following shows some examples of combining these techniques:

```
# 1. Simple check of two variables
if x != 0 or y != 0:
    print 'The point x,y is not on the x or y axis'

# 2. Checking for Pong paddle missing the ball for player 2
if ball_right_x > paddle_left_x and (ball_top_y > paddle_bottom_y or ball_bottom_y < pad
dle_top_y):
    player_1_score += 1

# The second example is a little long, we should simplify it
ball_above_paddle = ball_bottom_y < paddle_top_y
ball_below_paddle = ball_top_y > paddle_bottom_y
ball_right_paddle = ball_right_x > paddle_left_x

if ball_right_paddle and (ball_above_paddle or ball_below_paddle):
    player_1_score += 1
```

# Chained conditionals

Conditionals can also be **chained**

Chained conditionals use **elif** as an additonal check after the preceeding `if` predicate was False. For example

```
if x > 5:
    print 'x is greater than 5'
elif x < 5:
    print 'x is less than 5'
else:
    print 'x is equal to 5'
```

# Nested conditionals

Conditionals can also be **nested**

Nested conditionals occur inside of other conditionals and are indented over once more. When the code block is complete, they are unindented

```
if x > 5:
    print 'x is greater than 5'
    if x > 10:
        print '...it is also greater than 10'
    print 'Done evaluating the x > 10 block'
print 'Done evaluating the x > 5 block'
```

# Let's Develop It

Write a program that uses if statements to determine what to do given some user input

The code below is an example:

```
health = 100
print "A vicious warg is chasing you."
print "Options:"
print "1 - Hide in the cave."
print "2 - Climb a tree."
input_value = raw_input("Enter choice:")
if input_value == '1':
    print 'You hide in a cave.'
    print 'The warg finds you and injures your leg with its claws'
    health = health - 10
elif input_value == '2':
    print 'You climb a tree.'
    print 'The warg eventually looses interest and wanders off'
```

# Iteration

It is often useful to perform a task and to repeat the process until a certain point is reached.

The repeated execution of a set of statements is called **iteration**

One way to acheive this, is with the **while** loop.

```
x = 10
while x > 0:
    print x
    x = x - 1
print 'Done'
```

The while statement takes a predicate, and as long as it evaluates to True, the code block beneath it is repeated.

This creates a **loop**. Without the `x = x - 1` statement, this would be an **infinite loop**

# While loops

Consider the following example that uses iteration to derive a factorial

(A factorial of a number is equal to that number * every positive integer less than that number. E.g. The factorial of 4 is 4 * 3 * 2 * 1, which equals 24

```
input_value = raw_input('Enter a positive integer:')
n = int(input_value)
result = 1
while n > 1:
    result = result * n
    n = n - 1
print "The factorial of " + input_value + " is:"
print result
```

N.B. - This implementation does not work for negative numbers. Why?

# For loops

It is also useful to loop through a collection of elements, visiting each one to do some work, then stopping once all elements are processed.

This can be accomplished with a **for** loop

First, we need a collection. We create a **list** of numbers to loop over. This is called `numbers` in the following example

```
numbers = [1, 3, 8]
for number in numbers:
    print "The current number is:"
    print number
```

# For loops continued

Let's examine the example carefully

```
numbers = [1, 3, 8]
for number in numbers:
    print "The current number is:"
    print number
```

The for loop has three parts:

- The collection to loop over - numbers
- The name to give each element when the loop begins again - number
- The block of statements to execute with the element - The two print statements

# Break/Continue

- break - To exit early
- continue - to skip "this" iteration, and go to the next

```
numbers = [1, 2, 3]
for number in numbers:
    if number == 2:
        break
    else:
        print number

1

for number in numbers:
    if number == 2:
        continue
    else:
        print number

1
3
```

# Let's Develop It

- Write a program that obtains user input like the last program
- However, this program should not exit until the user types "quit".
- Hint: A loop should help you

# Functions

Also known as "procedures"

- A named unit of code that performs a specific task

When one uses a function, one makes a function **call**

We have already made a function call when using the type, int, or float functions

```
a = '3'
print type(a)
a = float(a)
```

# Function calls

```
a = 3
print type(a)
```

A function can take **arguments**

In the example above, the variable `a` is passed as an argument to the function `type`

Arguments can also be called **parameters**

```
# Some more function call examples

int('32')
str(32)
```

# Function definition

The following example is a **function definition**. This allows us to create our own functions

```
def print_plus_5(x):
    """Add 5 to any number"""
    print x + 5
```

The function definition has the following parts

- The **def** keyword signifies we are defining a function
- The name of the function being defined - `print_plus_5`
- The arguments in parentheses - `x`
- The document string. - `"""Add 5 to any number"""`
- The function **body**, which is a block of indented code that executes when the function is called. - `print x + 5`

# Function returns

A function can also **return** a value

To do this, one uses the **return** keyword

```
def plus_5(x):
    return x + 5

y = plus_5(4)
```

- This allows us to call a function to obtain a value for later use. (Not the same as printing the value)
- In this example, the function call `plus_5(4)` evaluates to 9, and y is set to this value
- To determine what a function will return, use the **substitution method**.
- If return is not used, the function returns **None**

# Functions with no arguments

A function does not have to take arguments, as in the following example:

```
def newline():
    print ''

newline()
# prints an empty line. Nothing is returned
```

This is useful when the function does some work but doesn't need any parameters. i.e. The function is intended to always do the same thing

# Functions with more than one argument

A function can also take more than one argument separated by commas. For example:

```
def find_rectangle_area(width, height):
    return width * height

area = find_rectangle_area(3, 4)
# area is set to the value 12
```

# Scope

The **scope** of a variable is the area of code in which a variable is still valid and can be used.

Variables defined within a function can not be used elsewhere.

```
def get_triangle_area(base, height):
    rect_area = base * height
    return rect_area / 2.0

triangle_area = get_triangle_area(10, 20)

print height
# NameError
print rect_area
# NameError
```

# Functions with Keyward Arguments

A function can also take keyword arguments separated by commas. For example:

```
def find_rectangle_area(width=1, height=0):
    return width * height

area = find_rectangle_area(height=4)
# area is set to the value 4
```

Keyword arguments are similar to arguments with the exception of the ability to set default values

Keyword arguments are by nature optional when calling a function.

# Scope

The **scope** of a variable is the area of code in which a variable is still valid and can be used.

Variables defined within a function can not be used elsewhere.

```
def get_triangle_area(base, height):
    rect_area = base * height
    return rect_area / 2.0

triangle_area = get_triangle_area(10, 20)

print height
# NameError
print rect_area
# NameError
```

# Import statements

The **import** statement allows us to use Python code that is defined in one file in a different file, or inside of the shell.

The **from** keyword allows us to only import parts of a Python file

```
# In knights.py
def shrubbery():
    print "I am a shrubber"

def ni():
    print "Ni!" * 3

# Run a Python shell in the same directory as knights.py and enter the following
import knights
knights.shrubbery()
knights.ni()

# or
from knights import shrubbery, ni
shrubbery()
ni()
```

# Let's Develop It

- Write a program that uses at least one function to solve a geometry problem
- Hint: You might use a loop that obtains user input, does the calculation, then prints the answer. As before, the user should be able to quit by entering "quit"
- Hint: You can import your function in the shell and call it with different parameters to test it out
- Hint: Download **geometry.py** and use it as an example

If you'd like to try something different or in addition, try the next slide...

# Let's Develop It

- Write a program that asks the user to guess a number between a given range, such as 1 to 10
- The program should give the user hints such as "too high" or "too low". Alternatively, the hints might be "warm" or "cold" depending on how close they are to the number
- The computer will need to have a random number for the user to guess:

```
#At the top of the file
from random import randint

# Use this line where you need to have a random number.
# (Hint, this is probably used before the user input loop)
random_number = randint(1, 10)
```

# Questions?