# Special Topics in Computer Science- CSC 4992

Basic Program Elements

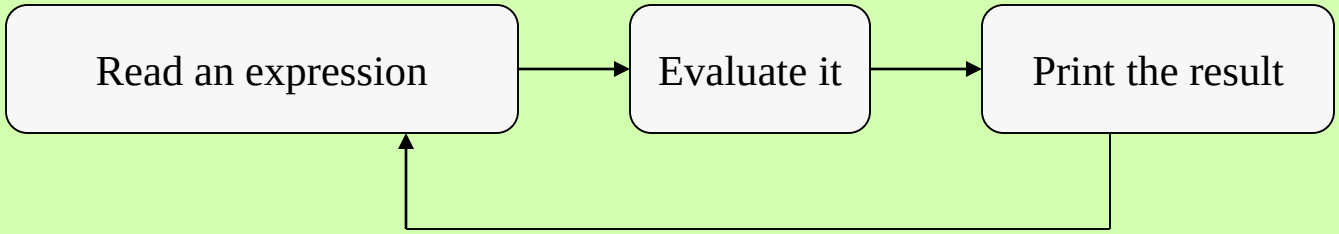
# Obtaining Python

- Python was invented by Guido van Rossum in 1992
- Python comes with most Unix-based computer systems
- Python for Windows or any other operating system can be downloaded from http://www.python.org/

# Developing Python Programs

- Can experiment with small pieces interactively in *shell mode*
- Can compose longer segments with an editor and run them in *script mode*

# Evaluating Python Expressions in Shell Mode (REPL)



# Basic Elements: Data

- Numbers
  - Integers: 3, 77
  - Floats: 3.14, .22
- Strings: ‘Hi there!’, “Hi there!”, ‘\n’
- Truth values (Booleans): True, False

# Basic Operations: Arithmetic

| Symbol | Meaning | Example |
|---|---|---|
| + | Addition or concatenation | x + y |
| - | Subtraction | x - y |
| * | Multiplication | x * y |
| / or // | Division | x / y or x // y |
| % | Remainder | x % y |
| ** | Exponentiation | x ** y |

# Built-In Functions

- A *function* is an operation that expects zero or more data values (arguments) from its user and computes and returns a single data value

- Examples:

```
abs(-5)

max(33, 66)
```

# Library Functions

- A *library* is a collection of resources, including functions, that can be *imported* for use in a program

- Example:

```
import math

math.sqrt(2)
```

# Variables and Assignment

- Variables are used to name data and other resources
- Instead of typing 3.14 for π, define a variable named **`pi`** to mean 3.14
- Assignment (=) is used to set (or reset) a variable to a value

```
pi = 3.14

34 * 34 * pi
```

- Variables make programs more readable and maintainable

# Library Variables

- Typically define standard constants, such as **`pi`** and **`e`**
- Example:

```
import math

3 * math.pi
```

- Or:

```
from math import pi

3 * pi
```

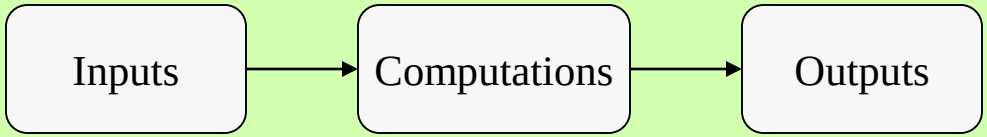
# Script Mode

- Longer programs can be edited and saved to a file and then run as *scripts* (synonymous with *programs*)
- *IDLE* is a script-mode development environment that can be used on any computer system

# Terminal-Based Programs

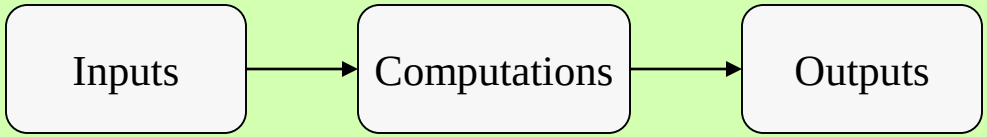
- A *terminal* allows a user to
  - run a program
  - view output as text on a screen or in a window
  - enter input as text from the keyboard
- Early computer systems were entirely terminal-based, modern systems have added a GUI (graphical user interface)

# Behavior of Terminal-Based Programs




- Prompt the user for some information
- Use that information to perform some computations
- Output the results

# Structure of Terminal-Based Programs




```
docstring

import statements

input statements

computation statements

output statements
```

Program code goes in a file with a **.py** extension.

# The docstring

```
but just you wait!
"""""
Author: Ken Lambert
This program does nothing
yet, "
```

Not evaluated, but used to document Python programs for other programmers

Should appear at the beginning of each file

Just as important as the evaluated code!!!

# `import` Statements

```
import math

print(math.pi)
print(math.sqrt(2))
```

Imports usually occur before the beginning of the executable program code

They make available resources from other Python modules

A *module* is just a file of Python code

# `import` Statements

```
from math import pi

print(pi)
```

Alternatively, one can import particular resources from a given module and then omit the module qualifier from the reference

# `import` Statements

```
from math import *

print(pi)
print(sqrt(2))
```

Or, one can import *all* the particular resources from a given module and then omit the module qualifier from all the references

# Input of Text

```
input('Enter your name: ')
```

The **`input`** function prints its string argument and waits for user input.

The function then returns the string of characters entered at the keyboard.

# Input of Numbers

```
int(input('Enter your age: '))
```

When an integer is expected, you must convert the input string to an **`int`**.

```
float(input('Enter your hourly wage: '))
```

When a real number (with a decimal point) is expected, you must convert the input string to a **`float`**.

# Simple Assignment Statements

```
name = input('Enter your name: ')

income = float(input('Enter your income: '))
```

The = operator evaluates the expression to its right and sets the variable to its left to the resulting value.

We use variables to retain data for further use.

***Note***: = does not mean *equals* in Python!

# Syntax Template for Simple Assignment

```
<variable> = <expression>
```

A *syntax template* expresses a grammar rule in a language.

```
area = math.pi * radius ** 2
century = 100
squarerootof2 = math.sqrt(2)
```

# More on Variables

```
firstname = input('Enter your first name: ')
```

Any variable can name any thing.

Variables must begin with a letter or the _ character.

They can contain any number of letters, digits, or _.

Variables cannot contain spaces.

Python is case-sensitive.  Use lowercase letters for now.

# Variable References

```
x = 10              # x begins as 10
x = x + 1           # x is reset to 11
y = y + x           # Error! Can't find value of y
```

When Python sees a variable in an expression, it must be able to look up its value.

If a variable has no established value, the program halts with an error message.

Variables are given values by assignment statements

# End of Line Comments

```
x = 10              # x begins as 10
x = x + 1           # x is reset to 11
y = y + x           # Error! Can't find value of y
```

**#** begins an end of line comment - Python ignores text from **#** to the end of line

# Evaluating Expressions

```
print(totalincome - deduction * rate)
```

```
print((totalincome - deduction) * rate)
```

```
print(10 + x * y ** 2)
```

Expressions are evaluated left to right, unless operator precedence overrides this order.

Use parentheses to override standard precedence when necessary.

# Mixed-Mode Arithmetic

```
print(5 * 100)
```

```
print(5 * 100.0)
```

The value of an expression depends on the *types* of its operands.

In general, two **`int`**s produce an **`int`**, whereas at least one **`float`** produces a **`float`**.

Exception: **`x / y`** always produces a **`float`**.

# Type Conversion Functions

```
str(3.72)          # Returns'3.72'

float('3.72')      # Returns 3.72

int(3.72)          # Returns 3

float(3)           # Returns 3.0
```

Each data type has a function to convert values of some other types to values of that type.

**`int`** *truncates* a **`float`** by removing the fractional part.

# Rounding and Precision

```
round(3.72)          # Returns 4

round(3.72, 1)       # Returns 3.7

round(3.729, 2)      # Returns 3.73
```

**round**'s optional second argument specifies the number of digits of precision in the fractional part

# Using Functions

```
round(3.72)          # returns 4

abs(-5)              # returns 5

math.sqrt(2)         # returns 1.4142135623730951
```

```
<function name>(<any arguments>)
```

A function can have one or more required *arguments* and/or some optional arguments

Arguments must be of the appropriate types

# Composing Expressions

```
squareofa = a ** 2
squareofb = b ** 2
sumofsquares = squareofa + squareofb
c = math.sqrt(sumofsquares)
print('The hypotenuse is', c)
```

Use assignment to name the results of computations

# Composing Expressions

```
squareofa = a ** 2
squareofb = b ** 2
sumofsquares = squareofa + squareofb
c = math.sqrt(sumofsquares)
print('The hypotenuse is', c)
```

Use assignment to name the results of computations

```
c = math.sqrt(a ** 2 + b ** 2)
print('The hypotenuse is', c)
```

Or just compose the expression and pass it as an argument to the function

# Getting the Directory of a Module

```
>>> import math

>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin',
'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e',
'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot',
'ldexp', 'log', 'log10', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']

>>>
```

The **`dir`** function returns a list of all of the named components in a module

# Getting Help on a Function

```
>>> import math

>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin',
'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e',
'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot',
'ldexp', 'log', 'log10', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']

>>> help(math.sqrt)
sqrt(x)

Return the square root of x.
```

# Output

```
print(3, 4)                          # displays 3 4

print(str(3) + str(4))               # displays 34

print('Hello there\nKen!')           # displays two lines

print('Hello there ', end='')        # displays one line
print('Ken')
```

**print** always ends output with a newline, unless its last argument is **end=''**