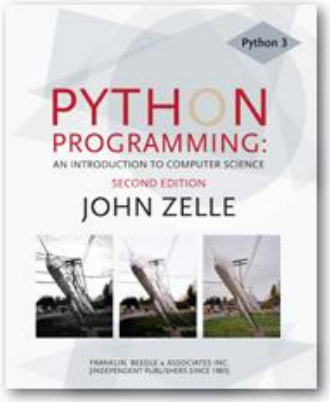
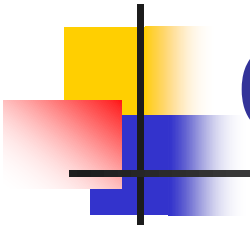
Python Programming: An Introduction to Computer Science

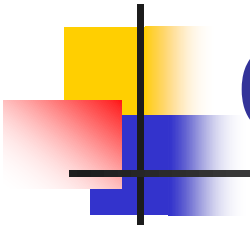


Chapter 3
Computing with Numbers



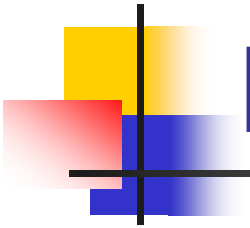
Objectives

- To understand the concept of data types.
- To be familiar with the basic numeric data types in Python.
- To understand the fundamental principles of how numbers are represented on a computer.



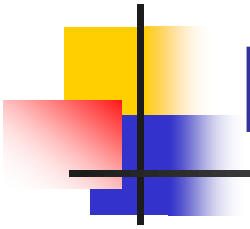
Objectives (cont.)

- To be able to use the Python math library.
- To understand the accumulator program pattern.
- To be able to read and write programs that process numerical data.



Numeric Data Types

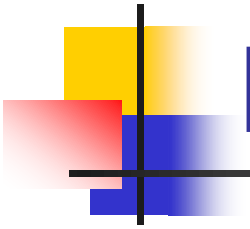
- The information that is stored and manipulated bu computers programs is referred to as *data*.
- There are two different kinds of numbers!
  - (5, 4, 3, 6) are whole numbers – they don't have a fractional part
  - (.25, .10, .05, .01) are decimal fractions



Numeric Data Types

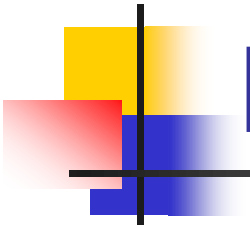
- Inside the computer, whole numbers and decimal fractions are represented quite differently!
- We say that decimal fractions and whole numbers are two different *data types*.

- The data type of an object determines what values it can have and what operations can be performed on it.



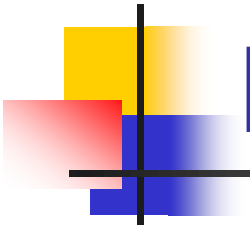
Numeric Data Types

- Whole numbers are represented using the *integer* (*int* for short) data type.
- These values can be positive or negative whole numbers.



Numeric Data Types

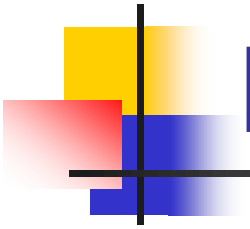
- Numbers that can have fractional parts are represented as *floating point* (or *float*) values.
- How can we tell which is which?
  - A numeric literal without a decimal point produces an int value
  - A literal that has a decimal point is represented by a float (even if the fractional part is 0)



Numeric Data Types

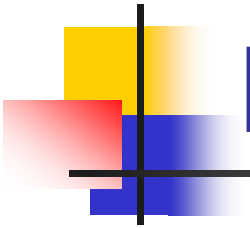
- Python has a special function to tell us the data type of any value.

```
>>> type(3)
<class 'int'>
>>> type(3.1)
<class 'float'>
>>> type(3.0)
<class 'float'>
>>> myInt = 32
>>> type(myInt)
<class 'int'>
>>>
```



Numeric Data Types

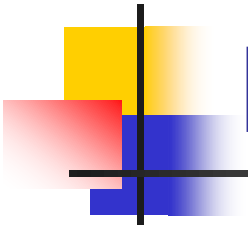
- Why do we need two number types?
  - Values that represent counts can't be fractional (you can't have 3 ½ quarters)
  - Most mathematical algorithms are very efficient with integers
  - The float type stores only an *approximation* to the real number being represented!
  - Since floats aren't exact, use an int whenever possible!



Numeric Data Types

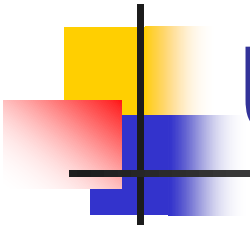
- Operations on ints produce ints, operations on floats produce floats (except for /).

```
>>> 3.0+4.0
7.0
>>> 3+4
7
>>> 3.0*4.0
12.0
>>> 3*4
12
>>> 10.0/3.0
3.3333333333333335
>>> 10/3
3.3333333333333335
>>> 10 // 3
3
>>> 10.0 // 3.0
3.0
```



Numeric Data Types

- Integer division produces a whole number.
- That's why 10//3 = 3!
- Think of it as 'gozinta', where 10//3 = 3 since 3 gozinta (goes into) 10 3 times (with a remainder of 1)
- 10%3 = 1 is the remainder of the integer division of 10 by 3.
- a = (a/b)(b) + (a%b)



Using the Math Library

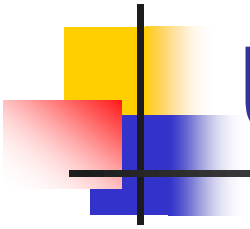
- Besides (+, -, *, /, //, **, %, abs), we have lots of other math functions available in a *math library*.
- A *library* is a module with some useful definitions/functions.

Using the Math Library

- Let's write a program to compute the roots of a quadratic equation!

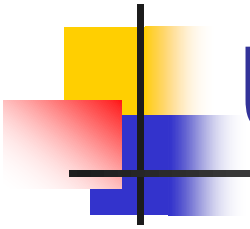
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- The only part of this we don't know how to do is find a square root… but it's in the math library!



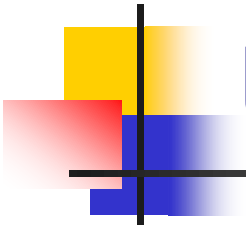
Using the Math Library

- To use a library, we need to make sure this line is in our program:
  import math
- Importing a library makes whatever functions are defined within it available to the program.



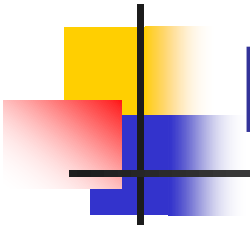
Using the Math Library

- To access the sqrt library routine, we need to access it as *math.sqrt(x)*.
- Using this dot notation tells Python to use the sqrt function found in the math library module.
- To calculate the root, you can do discRoot = math.sqrt(b*b – 4*a*c)



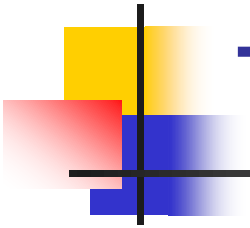
Using the Math Library

```
# quadratic.py
#    A program that computes the real roots of a quadratic equation.
#    Illustrates use of the math library.
#    Note: This program crashes if the equation has no real roots.

import math  # Makes the math library available.

def main():
    print("This program finds the real solutions to a quadratic")
    print()

    a, b, c = eval(input("Please enter the coefficients (a, b, c): "))

    discRoot = math.sqrt(b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)

    print()
    print("The solutions are:", root1, root2 )

main()
```



Math Library

- If a = 1, b = 2, c = 3, then we are trying to take the square root of a negative number!
- Using the sqrt function is more efficient than using **. How could you use ** to calculate a square root?

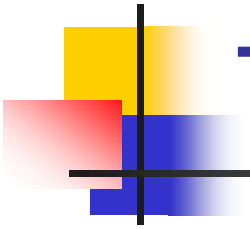


The Limits of Int

- What is 100!?

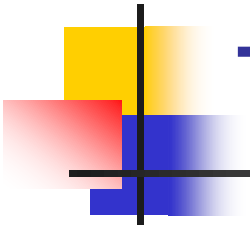
```
>>> main()
Please enter a whole number: 100
The factorial of 100 is
    933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
```

- Wow! That's a pretty big number!



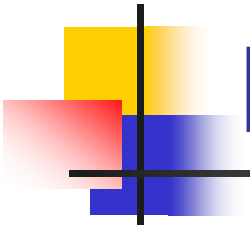
The Limits of Int

- What's going on?
  - While there are an infinite number of integers, there is a finite range of ints that can be represented.
  - This range depends on the number of *bits* a particular CPU uses to represent an integer value. Typical PCs use 32 bits.



The Limits of Int

- Typical PCs use 32 bits
- That means there are 2^{32} possible values, centered at 0.
- This range then is -2^{31} to 2^{31}-1. We need to subtract one from the top end to account for 0.
- But our 100! is much larger than this. How does it work?



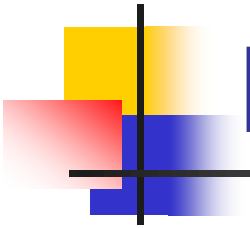
Handling Large Numbers

- Does switching to *float* data types get us around the limitations of *int*s?
- If we initialize the accumulator to 1.0, we get

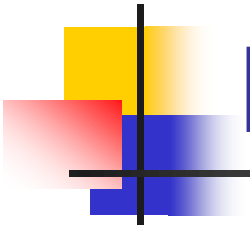
```
>>> main()
Please enter a whole number: 15
The factorial of 15 is 1.307674368e+012
```

- We no longer get an exact answer!

Handling Large Numbers: Long Int

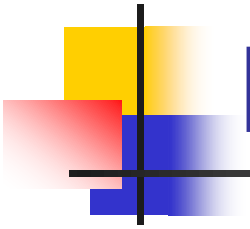


- Very large and very small numbers are expressed in *scientific* or *exponential notation*.
- 1.307674368e+012 means $1.307674368 * 10^{12}$
- Here the decimal needs to be moved right 12 decimal places to get the original number, but there are only 9 digits, so 3 digits of precision have been lost.



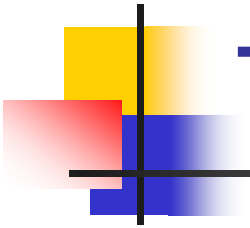
Handling Large Numbers

- Floats are approximations
- Floats allow us to represent a larger range of values, but with lower precision.
- Python has a solution, expanding ints!
- Python Ints are not a fixed size and expand to handle whatever value it holds.



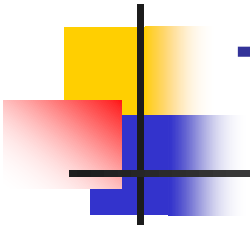
Handling Large Numbers

- Newer versions of Python automatically convert your ints to expanded form when they grow so large as to overflow.
- We get indefinitely large values (e.g. 100!) at the cost of speed and memory



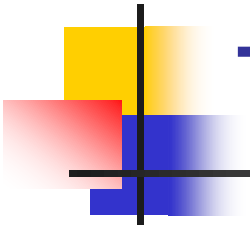
Type Conversions

- We know that combining an int with an int produces an int, and combining a float with a float produces a float.
- What happens when you mix an int and float in an expression?
  x = 5.0 + 2
- What do you think should happen?



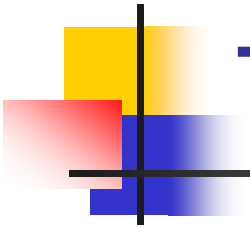
Type Conversions

- For Python to evaluate this expression, it must either convert 5.0 to 5 and do an integer addition, or convert 2 to 2.0 and do a floating point addition.
- Converting a float to an int will lose information
- Ints can be converted to floats by adding “.0”



Type Conversion

- In *mixed-typed expressions* Python will convert ints to floats.
- Sometimes we want to control the type conversion. This is called *explicit typing*.



Type Conversions

```
>>> float(22//5)
4.0
>>> int(4.5)
4
>>> int(3.9)
3
>>> round(3.9)
4
>>> round(3)
3
```

Assignment

- Create java program to calculate GPA by using information below:
- (input credit and grade by user)

Subject code	Credit	Grade
ICT101	3	4
ICT102	3	3.5
ICT103	3	3
ICT203	3	2.5
ICT306	3	3
ICT489	6	1
ICT211	3	1.5

GPA???