Outline

1. Variables and Assignments
   - assignment operators
   - types and arithmetic

2. Developing Python Programs
   - computing the yield of an investment
     from specification to implementation

3. Multiprecision Arithmetic
   - accuracy and precision
     arbitrary precision in Sage

4. Summary + Assignments

MCS 260 Lecture 4
Introduction to Computer Science
Jan Verschelde, 3 September 2008
Variables and Assignments
Developing Python Programs

1. Variables and Assignments
   assignment operators
types and arithmetic

2. Developing Python Programs
   computing the yield of an investment
   from specification to implementation

3. Multiprecision Arithmetic
   accuracy and precision
   arbitrary precision in Sage

4. Summary + Assignments
Suppose we invest $1,000 at an annual interest of 4%.
Then after one year the balance is

```python
>>> 1000*1.04
1040.0
```

If we want to continue our investment for another year, avoiding to recompute:

```python
>>> p = _
>>> p*1.04
1081.6000000000001
```

With the underscore operator `_` we recalled the result of the last calculation and assigned it to `p`. The effect is the same as `p = 1000*1.04`.
Suppose we invest $1,000 at an annual interest of 4%. Then after one year the balance is

```python
>>> 1000*1.04
1040.0
```

If we want to continue our investment for another year, avoiding to recompute:

```python
>>> p = _
>>> p*1.04
1081.6000000000001
```

With the underscore operator `_` we recalled the result of the last calculation and assigned it to `p`. The effect is the same as `p = 1000*1.04`. 
Suppose we invest $1,000 at an annual interest of 4%. Then after one year the balance is

```python
>>> 1000*1.04
1040.0
```

If we want to continue our investment for another year, avoiding to recompute:

```python
>>> p = _
>>> p*1.04
1081.6000000000001
```

With the underscore operator `_` we recalled the result of the last calculation and assigned it to `p`. The effect is the same as `p = 1000*1.04`. 
Assigning to Variables

how does an assignment work?

Stages in the execution of \( x = 3 \):

1. evaluate the right hand side of =
2. store the result of the evaluation in a register
3. compute the address of \( x \) and store it into the address register
4. copy the 3 from a register to the data register
5. put the value in data register at the memory location defined by the content of the address register
Assigning to Variables

how does an assignment work?

Stages in the execution of $x = 3$:

1. evaluate the right hand side of $=$
2. store the result of the evaluation in a register
3. compute the address of $x$ and store it into the address register
4. copy the $3$ from a register to the data register
5. put the value in data register at the memory location defined by the content of the address register
Assigning to Variables

how does an assignment work?

Stages in the execution of \( x = 3 \):

1. evaluate the right hand side of \( = \)
2. store the result of the evaluation in a register
3. compute the address of \( x \) and store it into the address register
4. copy the 3 from a register to the data register
5. put the value in data register at the memory location defined by the content of the address register
Assigning to Variables

how does an assignment work?

Stages in the execution of $x = 3$:

1. evaluate the right hand side of $=$
2. store the result of the evaluation in a register
3. compute the address of $x$
   and store it into the address register
4. copy the $3$ from a register to the data register
5. put the value in data register at the memory location
   defined by the content of the address register
Assigning to Variables
how does an assignment work?

Stages in the execution of $x = 3$:

1. evaluate the right hand side of $=$
2. store the result of the evaluation in a register
3. compute the address of $x$
   and store it into the address register
4. copy the 3 from a register to the data register
5. put the value in data register at the memory location defined by the content of the address register
Assigning to Variables
how does an assignment work?

Stages in the execution of $x = 3$:

1. evaluate the right hand side of $=$
2. store the result of the evaluation in a register
3. compute the address of $x$
   and store it into the address register
4. copy the 3 from a register to the data register
5. put the value in data register at the memory location
defined by the content of the address register
>>> x = 3
>>> id(x)
33568624

The machine view

In Python, the name x refers to the object 3.

In Python, the name x refers to the object 3.
>>> x = 3
>>> id(x)
33568624

The machine view

In Python, the name `x` refers to the object 3.
>>> x = 3
>>> id(x)
33568624

The machine view

In Python, the name \texttt{x} refers to the object 3.

In Python, the name \texttt{x} refers to the object 3.
Memory Management
implicit and explicit

- **Automatic allocation of memory for each object.**
- Garbage collection frees space for unused variables.
- With `del` we may explicitly free space:

  ```python
  >>> x = 3
  >>> y = x
  >>> id(x)
  33568624
  >>> id(y)
  33568624
  
  We see that both `x` and `y` refer to the same object.
  ```

  ```python
  >>> del(x)
  >>> y
  3
  
  After deleting `x`, the object is still accessible via `y`. Only after `del(y)` will memory space be released.
Memory Management

implicit and explicit

- Automatic allocation of memory for each object.
- Garbage collection frees space for unused variables.
- With `del` we may explicitly free space:

```
>>> x = 3
>>> y = x
>>> id(x)
33568624
>>> id(y)
33568624
```
We see that both `x` and `y` refer to the same object.
```
>>> del(x)
>>> y
3
```
After deleting `x`, the object is still accessible via `y`. Only after `del(y)` will memory space be released.
Memory Management

implicit and explicit

- Automatic allocation of memory for each object.
- Garbage collection frees space for unused variables.
- With `del` we may explicitly free space:

```python
>>> x = 3
>>> y = x
>>> id(x)
33568624
>>> id(y)
33568624
```

We see that both `x` and `y` refer to the same object.

```python
>>> del(x)
>>> y
3
```

After deleting `x`, the object is still accessible via `y`. Only after `del(y)` will memory space be released.
Memory Management
implicit and explicit

- Automatic allocation of memory for each object.
- Garbage collection frees space for unused variables.
- With `del` we may explicitly free space:

```
>>> x = 3
>>> y = x
>>> id(x)
33568624
>>> id(y)
33568624

We see that both `x` and `y` refer to the same object.
```

```
>>> del(x)
>>> y
3

After deleting `x`, the object is still accessible via `y`. Only after `del(y)` will memory space be released.
```
Assignment Operators

making updates to variables

Note that $x = 3 \neq x == 3!$

Often assignments are updates to variables, therefore:

<table>
<thead>
<tr>
<th>operator</th>
<th>example</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=$</td>
<td>$x = y$</td>
<td>afterwards: $x == y$</td>
</tr>
<tr>
<td>$+= $</td>
<td>$x += y$</td>
<td>equivalent to $x = x + y$</td>
</tr>
<tr>
<td>$-= $</td>
<td>$x -= y$</td>
<td>equivalent to $x = x - y$</td>
</tr>
<tr>
<td>$\ast=$</td>
<td>$x \ast = y$</td>
<td>equivalent to $x = x \ast y$</td>
</tr>
<tr>
<td>$/=$</td>
<td>$x /= y$</td>
<td>equivalent to $x = x / y$</td>
</tr>
<tr>
<td>$%=$</td>
<td>$x % = y$</td>
<td>equivalent to $x = x % y$</td>
</tr>
</tbody>
</table>

The $x += y$ avoids a duplicate address calculation for $x$.

Multiple assignments $a = b = 1$ useful as initializations.
Assignment Operators
making updates to variables

Note that $x = 3 \neq x == 3$!

Often assignments are updates to variables, therefore:

<table>
<thead>
<tr>
<th>operator</th>
<th>example</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=$</td>
<td>$x = y$</td>
<td>afterwards: $x == y$</td>
</tr>
<tr>
<td>$+=</td>
<td>$x += y$</td>
<td>equivalent to $x = x + y$</td>
</tr>
<tr>
<td>$-=</td>
<td>$x -= y$</td>
<td>equivalent to $x = x - y$</td>
</tr>
<tr>
<td>$*=</td>
<td>$x *= y$</td>
<td>equivalent to $x = x * y$</td>
</tr>
<tr>
<td>$/=$</td>
<td>$x /= y$</td>
<td>equivalent to $x = x / y$</td>
</tr>
<tr>
<td>$%=$</td>
<td>$x %= y$</td>
<td>equivalent to $x = x % y$</td>
</tr>
</tbody>
</table>

The $x += y$ avoids a duplicate address calculation for $x$.

Multiple assignments $a = b = 1$ useful as initializations.
Assignment Operators

making updates to variables

Note that $x = 3 \neq x == 3$!

Often assignments are updates to variables, therefore:

<table>
<thead>
<tr>
<th>operator</th>
<th>example</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=$</td>
<td>$x = y$</td>
<td>afterwards: $x == y$</td>
</tr>
<tr>
<td>$+= $</td>
<td>$x += y$</td>
<td>equivalent to $x = x + y$</td>
</tr>
<tr>
<td>$-= $</td>
<td>$x -= y$</td>
<td>equivalent to $x = x - y$</td>
</tr>
<tr>
<td>$*=$</td>
<td>$x *= y$</td>
<td>equivalent to $x = x * y$</td>
</tr>
<tr>
<td>$/=$</td>
<td>$x /= y$</td>
<td>equivalent to $x = x / y$</td>
</tr>
<tr>
<td>$%=$</td>
<td>$x %= y$</td>
<td>equivalent to $x = x % y$</td>
</tr>
</tbody>
</table>

The $x += y$ avoids a duplicate address calculation for $x$.

Multiple assignments $a = b = 1$ useful as initializations.
Assignment Operators

document updates to variables

Note that $x = 3 \neq x == 3!$

Often assignments are updates to variables, therefore:

<table>
<thead>
<tr>
<th>operator</th>
<th>example</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=$</td>
<td>$x = y$</td>
<td>afterwards: $x == y$</td>
</tr>
<tr>
<td>$+= $</td>
<td>$x += y$</td>
<td>equivalent to $x = x + y$</td>
</tr>
<tr>
<td>$-= $</td>
<td>$x -= y$</td>
<td>equivalent to $x = x - y$</td>
</tr>
<tr>
<td>$*=$</td>
<td>$x *= y$</td>
<td>equivalent to $x = x * y$</td>
</tr>
<tr>
<td>$/=$</td>
<td>$x /= y$</td>
<td>equivalent to $x = x / y$</td>
</tr>
<tr>
<td>$%=$</td>
<td>$x %= y$</td>
<td>equivalent to $x = x % y$</td>
</tr>
</tbody>
</table>

The $x += y$ avoids a duplicate address calculation for $x$. Multiple assignments $a = b = 1$ useful as initializations.
Variables and Assignments
Developing Python Programs

1. Variables and Assignments
   assignment operators
types and arithmetic

2. Developing Python Programs
   computing the yield of an investment
   from specification to implementation

3. Multiprecision Arithmetic
   accuracy and precision
   arbitrary precision in Sage

4. Summary + Assignments
Types of Variables
every variable has a type

- Python uses *dynamic typing*: the type of a variable is determined automatically at runtime.
  \[ \text{type(name)} \text{ returns type of the variable name} \]

- The type of a variable determines
  ① what kind of operations are available
  ② the outcome of those operations, e.g.: \( 1/3 \neq 1.0/3 \)

- We have elementary and composite types:
  a character is an elementary type, while a string is a sequence of characters, a composite type.

- Elementary types: int, float, char, and boolean.
Types of Variables

every variable has a type

- **Python uses** *dynamic typing*: the type of a variable is determined automatically at runtime.
  
  \[
  \text{type(name)} \text{ returns type of the variable } \text{name}
  \]

- **The type of a variable determines**
  
  1. what kind of operations are available
  2. the outcome of those operations, e.g.: \(1/3 \neq 1.0/3\)

- We have elementary and composite types:
  
  a character is an elementary type, while a string is a sequence of characters, a composite type.

- Elementary types: int, float, char, and boolean.
Types of Variables

every variable has a type

- Python uses *dynamic typing*: the type of a variable is determined automatically at runtime. 
  
  ```python
type(name)
  ```
  
  returns type of the variable *name*

- The type of a variable determines
  
  1. what kind of operations are available
  2. the outcome of those operations, e.g.: \(1/3 \neq 1.0/3\)

- We have elementary and composite types:
  
  a character is an elementary type, while a string is a sequence of characters, a composite type.

- Elementary types: int, float, char, and boolean.
Types of Variables

every variable has a type

- Python uses *dynamic typing*: the type of a variable is determined automatically at runtime.
  
  \[
  \text{type(name)} \text{ returns type of the variable name}
  \]

- The type of a variable determines
  1. what kind of operations are available
  2. the outcome of those operations, e.g.: \( 1/3 \neq 1.0/3 \)

- We have elementary and composite types:
  
  a character is an elementary type, while a string is a sequence of characters, a composite type.

- Elementary types: int, float, char, and boolean.
Types of Variables

every variable has a type

- Python uses *dynamic typing*: the type of a variable is determined automatically at runtime. `type(name)` returns type of the variable `name`

- The type of a variable determines
  1. what kind of operations are available
  2. the outcome of those operations, e.g.: \( \frac{1}{3} \neq \frac{1.0}{3} \)

- We have elementary and composite types:
  a character is an elementary type, while a string is a sequence of characters, a composite type.

- Elementary types: int, float, char, and boolean.
Types of Variables
every variable has a type

- Python uses *dynamic typing*: the type of a variable is determined automatically at runtime. `type(name)` returns type of the variable `name`.

- The type of a variable determines
  1. what kind of operations are available
  2. the outcome of those operations, e.g.: `1/3 \neq 1.0/3`

- We have elementary and composite types:
  a character is an elementary type, while a string is a sequence of characters, a composite type.

- Elementary types: int, float, char, and boolean.
String Concatenation
adding strings together

>>> x = 'a'

>>> type(x)
<type 'str'>

A character is a string of length one. Python has no separate type for a character.

We can apply the + operator to strings:

>>> a = 'hello'

>>> b = 'world'

>>> c = a + '' + b

>>> c
'hello world'

On strings, the + operator concatenates. The outcome of operations depends on the types of the operands.
String Concatenation
adding strings together

>>> x = 'a'
>>> type(x)
<type 'str'>

A character is a string of length one.
Python has no separate type for a character.

We can apply the + operator to strings:

>>> a = 'hello'
>>> b = 'world'
>>> c = a + ' ' + b
>>> c
'hello world'

On strings, the + operator concatenates. The outcome of operations depends on the types of the operands.
Integer Numbers
machine integers and long integers

- The computer uses 32 bits to represent integers. First bit is sign bit: 0 = +, 1 = −.

- The largest machine int is $+2^{31} - 1$:
  
  ```python
  >>> 2**30 + (2**30 - 1)
  2147483647
  >>> _ + 1
  2147483648L
  
  The smallest machine int is $-2^{31}$:
  
  ```python
  >>> -2**30 - 2**30
  -2147483648
  >>> _ - 1
  -2147483649L
  ```

- Use of long integers avoids over and underflow.
Integer Numbers
machine integers and long integers

- The computer uses 32 bits to represent integers. First bit is sign bit: 0 = +, 1 = −.
- The largest machine int is $+2^{31} - 1$:

  ```python
  >>> 2**30 + (2**30 - 1)
  2147483647
  >>> _ + 1
  2147483648L
  ```

- The smallest machine int is $-2^{31}$:

  ```python
  >>> -2**30 - 2**30
  -2147483648
  >>> _ - 1
  -2147483649L
  ```

- Use of long integers avoids over and underflow.
Integer Numbers

machine integers and long integers

- The computer uses 32 bits to represent integers. First bit is sign bit: 0 = +, 1 = −.
- The largest machine int is $+2^{31} - 1$:
  ```python
  >>> 2**30 + (2**30 - 1)
  2147483647
  >>> _ + 1
  2147483648L
  ```
- The smallest machine int is $-2^{31}$:
  ```python
  >>> -2**30 - 2**30
  -2147483648
  >>> _ - 1
  -2147483649L
  ```
- Use of long integers avoids over and underflow.
The computer uses 32 bits to represent integers. First bit is sign bit: $0 = +$, $1 = -$.

The largest machine int is $+2^{31} - 1$:

```python
>>> 2**30 + (2**30 - 1)
2147483647
>>> _ + 1
2147483648L
```

The smallest machine int is $-2^{31}$:

```python
>>> -2**30 - 2**30
-2147483648
>>> _ - 1
-2147483649L
```

Use of long integers avoids over and underflow.
Floating-Point Numbers
approximate real computations

- For large numbers, magnitude matters most. In many applications, the input data is approximate.

- A floating-point consists of a fraction $f$ and an exponent $e$: $x = f \times b^e$, where the base $b$ is mostly 2. Normal form of $x$: adjust $e$ so $f$’s leading bit $\neq 0$. $b = 10$, $#f = 2$, $100 = 10^2 = .10e+3$.

- The typical machine float uses 64 bits: 1 sign bit, 11 bits for the exponent, and 52 bits for the fraction.

- The machine precision is the largest positive number we can add to 1.0 without making a difference.

  >>> 1.0+2.0**(-52)
  1.0000000000000002
  >>> 1.0+2.0**(-53)
  1.0

- Largest float is $2.0**(2**10-1)$.
Floating-Point Numbers
approximate real computations

- For large numbers, magnitude matters most. In many applications, the input data is approximate.
- A floating-point consists of a fraction $f$ and an exponent $e$: $x = f \times b^e$, where the base $b$ is mostly 2. Normal form of $x$: adjust $e$ so $f$’s leading bit $\neq 0$. $b = 10$, $\#f = 2$, $100 = 10^2 = .10_{e+3}$.
- The typical machine float uses 64 bits: 1 sign bit, 11 bits for the exponent, and 52 bits for the fraction.
- The **machine precision** is the largest positive number we can add to 1.0 without making a difference.

```python
>>> 1.0+2.0**(-52)
1.0000000000000002
>>> 1.0+2.0**(-53)
1.0
```

- Largest float is $2.0^{(2**10-1)}$. 
Floating-Point Numbers
approximate real computations

- For large numbers, magnitude matters most. In many applications, the input data is approximate.
- A floating-point consists of a fraction \( f \) and an exponent \( e \): \( x = f \times b^e \), where the base \( b \) is mostly 2. Normal form of \( x \): adjust \( e \) so \( f \)’s leading bit \( \neq 0 \).
  \( b = 10, \#f = 2, 100 = 10^2 = .10 \times 10^3 \).
- The typical machine float uses 64 bits: 1 sign bit, 11 bits for the exponent, and 52 bits for the fraction.
- The **machine precision** is the largest positive number we can add to 1.0 without making a difference.

```python
>>> 1.0+2.0**(-52)
1.0000000000000002
>>> 1.0+2.0**(-53)
1.0
```
- Largest float is \( 2.0^{2^{10}-1} \).
Floating-Point Numbers
approximate real computations

- For large numbers, magnitude matters most. In many applications, the input data is approximate.
- A floating-point consists of a fraction \( f \) and an exponent \( e \): \( x = f \times b^e \), where the base \( b \) is mostly 2. Normal form of \( x \): adjust \( e \) so \( f \)'s leading bit \( \neq 0 \). \( b = 10, \#f = 2, 100 = 10^2 = .10e+3 \).
- The typical machine float uses 64 bits: 1 sign bit, 11 bits for the exponent, and 52 bits for the fraction.
- The **machine precision** is the largest positive number we can add to 1.0 without making a difference.
  
  ```
  >>> 1.0+2.0**(-52)
  1.0000000000000002
  >>> 1.0+2.0**(-53)
  1.0
  ```

  - Largest float is \( 2.0^{(2^{10}-1)} \).
Truncation and Rounding
converting floats to integers

```python
>>> x = 3.7
>>> int(x)  # drops all digits after .
3
>>> round(x)
4.0
>>> type(round(x))
'<type 'float'>
```

By default, `round` returns a floating-point number rounded with 0 digits after the decimal point.

```python
>>> import math
>>> p = math.pi
>>> p
3.1415926535897931
>>> round(p, 2)
3.1400000000000001
```
Truncation and Rounding
converting floats to integers

```python
>>> x = 3.7
>>> int(x)  # drops all digits after .
3
>>> round(x)
4.0
>>> type(round(x))
<type 'float'>
```

By default, `round` returns a floating-point number rounded with 0 digits after the decimal point.

```python
>>> import math
>>> p = math.pi
>>> p
3.1415926535897931
>>> round(p, 2)
3.1400000000000001
```
Boolean

store outcome of logical expressions

>>> x = 3
>>> x == 3
True
>>> x == 4
False

For complicated logical expressions, we may want to save its outcome (Python session continued):

>>> b = _
>>> type(b)
<type 'bool'>
Variables and Assignments
Developing Python Programs

1 Variables and Assignments
   assignment operators
types and arithmetic

2 Developing Python Programs
   computing the yield of an investment
from specification to implementation

3 Multiprecision Arithmetic
   accuracy and precision
   arbitrary precision in Sage

4 Summary + Assignments
the Yield of an Investment
program specification

Suppose we want an interactive program to compute the end balance and yield of an investment after some years.

**Input** principal, how much will be invested; annual interest rate (given as a percentage); number of years the investment will run.

**Output** the ending balance of the investment; the yield (balance – principal).

This Input/Output description can be regarded as part of the program specification, describing unambiguously what the program does.
the Yield of an Investment
program specification

Suppose we want an interactive program to compute the end balance and yield of an investment after some years.

**Input** principal, how much will be invested; annual interest rate (given as a percentage); number of years the investment will run.

**Output** the ending balance of the investment; the yield (balance – principal).

This Input/Output description can be regarded as part of the **program specification**, describing unambiguously what the program does.
Suppose we want an interactive program to compute the end balance and yield of an investment after some years.

**Input** principal, how much will be invested; annual interest rate (given as a percentage); number of years the investment will run.

**Output** the ending balance of the investment; the yield (balance — principal).

This Input/Output description can be regarded as part of the program specification, describing unambiguously what the program does.
Suppose we want an interactive program to compute the end balance and yield of an investment after some years.

**Input** principal, how much will be invested; annual interest rate (given as a percentage); number of years the investment will run.

**Output** the ending balance of the investment; the yield (balance − principal).

This Input/Output description can be regarded as part of the program specification, describing unambiguously what the program does.
Suppose we save the program as `yieldbal.py`.

Then running the program could go as

```
$ python yieldbal.py
computation of yield and balance
Give the principal : 1891.50
Give the annual interest rate : 3.12
Give the number of years : 4
Investing $1891.50 at 3.12% for 4 years gives $2138.84 as balance and $247.34 as yield.
```

Suppose we save the program as `yieldbal.py`.

Then running the program could go as

```
$ python yieldbal.py
computation of yield and balance
Give the principal : 1891.50
Give the annual interest rate : 3.12
Give the number of years : 4
Investing $1891.50 at 3.12% for 4 years
gives $2138.84 as balance and $247.34 as yield.
$
Suppose we save the program as `yieldbal.py`.

Then running the program could go as

```bash
$ python yieldbal.py
computation of yield and balance
Give the principal : 1891.50
Give the annual interest rate : 3.12
Give the number of years : 4
Investing $1891.50 at 3.12% for 4 years
gives $2138.84 as balance and $247.34 as yield.
$
Specify I/O Formats

Suppose we save the program as `yieldbal.py`.

Then running the program could go as

```
$ python yieldbal.py
computation of yield and balance
Give the principal : 1891.50
Give the annual interest rate : 3.12
Give the number of years : 4
Investing $1891.50 at 3.12\% for 4 years gives $2138.84 as balance and $247.34 as yield.
```


Specifying I/O Formats
a more detailed specification

Suppose we save the program as `yieldbal.py`.

Then running the program could go as

```
$ python yieldbal.py
computation of yield and balance
Give the principal : 1891.50
Give the annual interest rate : 3.12
Give the number of years : 4
Investing $1891.50 at 3.12% for 4 years
gives $2138.84 as balance and $247.34 as yield.
```

$
Variables and Assignments
Developing Python Programs

1. Variables and Assignments
   assignment operators
types and arithmetic

2. Developing Python Programs
   computing the yield of an investment
   from specification to implementation

3. Multiprecision Arithmetic
   accuracy and precision
   arbitrary precision in Sage

4. Summary + Assignments
Specification → Implementation

a top down incremental approach to programming

Several stages to develop the program:

- select an algorithm to solve the problem
  balance \( B = P(1 + r/100)^n \) and yield \( y = B - P \),
  for principal \( P \), interest rate \( r \) and \( n \) years.

- write the outline in pseudo code
  1. read input and convert strings to numbers
  2. confirm the input of the user
  3. compute balance and yield
  4. show the results to the user

- choose names for the variables
  single letters \( P, r, n, B, y \) often not so meaningful

- type in actual Python code and test
Specification → Implementation
a top down incremental approach to programming

Several stages to develop the program:

- select an algorithm to solve the problem balance \( B = P(1 + r/100)^n \) and yield \( y = B - P \), for principal \( P \), interest rate \( r \) and \( n \) years.

- write the outline in pseudo code
  1. read input and convert strings to numbers
  2. confirm the input of the user
  3. compute balance and yield
  4. show the results to the user

- choose names for the variables
  single letters \( P, r, n, B, y \) often not so meaningful

- type in actual Python code and test
Specification → Implementation

a top down incremental approach to programming

Several stages to develop the program:

- select an algorithm to solve the problem balance $B = P(1 + r/100)^n$ and yield $y = B - P$, for principal $P$, interest rate $r$ and $n$ years.

- write the outline in pseudo code
  1. read input and convert strings to numbers
  2. confirm the input of the user
  3. compute balance and yield
  4. show the results to the user

- choose names for the variables
  single letters $P$, $r$, $n$, $B$, $y$ often not so meaningful

- type in actual Python code and test
Specification → Implementation

a top down incremental approach to programming

Several stages to develop the program:

- select an algorithm to solve the problem
  balance $B = P(1 + r/100)^n$ and yield $y = B - P$, for principal $P$, interest rate $r$ and $n$ years.
- write the outline in pseudo code
  1. read input and convert strings to numbers
  2. confirm the input of the user
  3. compute balance and yield
  4. show the results to the user
- choose names for the variables
  single letters $P$, $r$, $n$, $B$, $y$ often not so meaningful
- type in actual Python code and test
Specification → Implementation

a top down incremental approach to programming

Several stages to develop the program:

- select an algorithm to solve the problem
  balance \( B = P(1 + r/100)^n \) and yield \( y = B - P \),
  for principal \( P \), interest rate \( r \) and \( n \) years.

- write the outline in pseudo code
  1. read input and convert strings to numbers
  2. confirm the input of the user
  3. compute balance and yield
  4. show the results to the user

- choose names for the variables
  single letters \( P \), \( r \), \( n \), \( B \), \( y \) often not so meaningful

- type in actual Python code and test
Specification → Implementation

a top down incremental approach to programming

Several stages to develop the program:

- select an algorithm to solve the problem balance $B = P(1 + r/100)^n$ and yield $y = B - P$, for principal $P$, interest rate $r$ and $n$ years.
- write the outline in pseudo code
  1. read input and convert strings to numbers
  2. confirm the input of the user
  3. compute balance and yield
  4. show the results to the user
- choose names for the variables
  single letters $P$, $r$, $n$, $B$, $y$ often not so meaningful
- type in actual Python code and test
Specification → Implementation

a top down incremental approach to programming

Several stages to develop the program:

- select an algorithm to solve the problem
  balance \( B = P(1 + r/100)^n \) and yield \( y = B - P \),
  for principal \( P \), interest rate \( r \) and \( n \) years.

- write the outline in pseudo code
  1. read input and convert strings to numbers
  2. confirm the input of the user
  3. compute balance and yield
  4. show the results to the user

- choose names for the variables
  single letters \( P, r, n, B, y \) often not so meaningful

- type in actual Python code and test
Specifying → Implementation

a top down incremental approach to programming

Several stages to develop the program:

- select an algorithm to solve the problem balance \( B = P(1 + r/100)^n \) and yield \( y = B - P \), for principal \( P \), interest rate \( r \) and \( n \) years.
- write the outline in pseudo code
  1. read input and convert strings to numbers
  2. confirm the input of the user
  3. compute balance and yield
  4. show the results to the user
- choose names for the variables
  single letters \( P, r, n, B, y \) often not so meaningful
- type in actual Python code and test
Specification → Implementation

a top down incremental approach to programming

Several stages to develop the program:

- select an algorithm to solve the problem balance $B = P(1 + r/100)^n$ and yield $y = B - P$, for principal $P$, interest rate $r$ and $n$ years.
- write the outline in pseudo code
  1. read input and convert strings to numbers
  2. confirm the input of the user
  3. compute balance and yield
  4. show the results to the user
- choose names for the variables
  single letters $P$, $r$, $n$, $B$, $y$ often not so meaningful
- type in actual Python code and test
Specification → Implementation
a top down incremental approach to programming

Several stages to develop the program:

- select an algorithm to solve the problem balance $B = P(1 + r/100)^n$ and yield $y = B - P$, for principal $P$, interest rate $r$ and $n$ years.
- write the outline in pseudo code
  1. read input and convert strings to numbers
  2. confirm the input of the user
  3. compute balance and yield
  4. show the results to the user
- choose names for the variables
  single letters $P$, $r$, $n$, $B$, $y$ often not so meaningful
- type in actual Python code and test
Flowchart of a Formula

\[ B = P(1 + r/100)^n \]

1. prompt user for principal
2. prompt user for rate
3. prompt user for years
4. print principal, rate, years
5. \[ balance = principal * (1 + rate/100)^{years} \]
6. \[ yield = balance - principal \]
7. print balance, yield
Variables and Assignments

Developing Python Programs
computing the yield of an investment from specification to implementation

Multiprecision Arithmetic
accuracy and precision arbitrary precision in Sage

Summary + Assignments

Flowchart of a Formula

\[ B = P(1 + r/100)^n \]

- Prompt user for principal
- Prompt user for rate
- Prompt user for years
- Print principal, rate, years
- \[ \text{balance} = \text{principal} \times (1 + \text{rate}/100)^\text{years} \]
- \[ \text{yield} = \text{balance} - \text{principal} \]
- Print balance, yield
Developing Python Programs
computing the yield of an investment
from specification to implementation

Multiprecision Arithmetic
accuracy and precision
arbitrary precision in Sage

Summary + Assignments

Variables and Assignments
assignment operators
types and arithmetic

Flowchart of a Formula

\[ B = P(1 + r/100)^n \]

prompt user for principal

prompt user for rate

prompt user for years

print principal, rate, years

balance = principal*(1+rate/100)**years

yield = balance - principal

print balance, yield
0. Structure of the Program

We write the program specification and pseudo code for the outline of the program as comments:

```python
# L-4 MCS 260 Wed 3 Sep 2008 yield and balance
#
# This program prompts the user for 3 inputs:
#   (1) principal, the amount to invest,
#   (2) annual interest rate (as a percentage),
#   (3) and number of years.
# Types of (1), (2) is float, (3) is integer.
# The output of the program consists of
#   (1) the ending balance and
#   (2) the yield of the investment.
#
# welcome message, read input and convert
# confirm the user input
# calculate balance and yield
# show the results to screen
```
0. Structure of the Program

We write the program specification and pseudo code for the outline of the program as comments:

```python
# L-4 MCS 260 Wed 3 Sep 2008 yield and balance
#
# This program prompts the user for 3 inputs:
# (1) principal, the amount to invest,
# (2) annual interest rate (as a percentage),
# (3) and number of years.
# Types of (1), (2) is float, (3) is integer.
# The output of the program consists of
# (1) the ending balance and
# (2) the yield of the investment.
#
# welcome message, read input and convert
# confirm the user input
# calculate balance and yield
# show the results to screen
```
0. Structure of the Program

We write the program specification and pseudo code for the outline of the program as comments:

```python
# L-4 MCS 260 Wed 3 Sep 2008 yield and balance
#
# This program prompts the user for 3 inputs:
#  (1) principal, the amount to invest,
#  (2) annual interest rate (as a percentage),
#  (3) and number of years.
# Types of (1), (2) is float, (3) is integer.
# The output of the program consists of
#  (1) the ending balance and
#  (2) the yield of the investment.
#
# welcome message, read input and convert
# confirm the user input
# calculate balance and yield
# show the results to screen
```
0. Structure of the Program

We write the program specification and pseudo code for the outline of the program as comments:

```
# L-4 MCS 260 Wed 3 Sep 2008 yield and balance
#
# This program prompts the user for 3 inputs:
#   (1) principal, the amount to invest,
#   (2) annual interest rate (as a percentage),
#   (3) and number of years.
# Types of (1), (2) is float, (3) is integer.
# The output of the program consists of
#   (1) the ending balance and
#   (2) the yield of the investment.
#
# welcome message, read input and convert
# confirm the user input
# calculate balance and yield
# show the results to screen
```
1. Read Input from User
assuming all user input is correct

All output of `raw_input` is assigned to `user_input`. We calculate with `principal`, `interest` and `years`.

```python
# welcome message, read input and convert
print 'computation of yield and balance'
user_input = raw_input('Give the principal : ')
principal = float(user_input)
user_input = raw_input('Give the annual interest rate : ')
interest = float(user_input)
user_input = raw_input('Give the number of years : ')
years = int(user_input)
```

The backslash `\` allows to continue long lines of code.
1. Read Input from User
assuming all user input is correct

All output of `raw_input` is assigned to `user_input`. We calculate with `principal`, `interest` and `years`.

```python
# welcome message, read input and convert
print('computation of yield and balance')
user_input = raw_input('Give the principal : ')
principal = float(user_input)
user_input = raw_input('Give the annual interest rate : ')
interest = float(user_input)
user_input = raw_input('Give the number of years : ')
years = int(user_input)
```

The backslash \ allows to continue long lines of code.
1. Read Input from User
assuming all user input is correct

All output of `raw_input` is assigned to `user_input`. We calculate with `principal`, `interest` and `years`.

```python
# welcome message, read input and convert
print 'computation of yield and balance'
user_input = raw_input('Give the principal : ')
principal = float(user_input)
user_input = raw_input('Give the annual interest rate : ')
interest = float(user_input)
user_input = raw_input('Give the number of years : ')
years = int(user_input)
```

The backslash `\` allows to continue long lines of code.
1. Read Input from User

assuming all user input is correct

All output of `raw_input` is assigned to `user_input`. We calculate with `principal`, `interest` and `years`.

```python
# welcome message, read input and convert
print 'computation of yield and balance'
user_input = raw_input('Give the principal : ')
principal = float(user_input)
user_input = raw_input('Give the annual interest rate : ')
interest = float(user_input)
user_input = raw_input('Give the number of years : ')
years = int(user_input)
```

The backslash `\` allows to continue long lines of code.
2. Confirm the User Input

using format operators and string concatenations

The `%` operator is used to confirm user input:
'`%2.2f'` to show floats with two digits after .
'`%d'` to convert an integer into a string.

Strings are stored in `confirm_principal`, `confirm_interest`, and `confirm_years`.

```python
# confirm the user input
confirm_principal = 'Investing ' + '$%.2f' % principal
confirm_interest = ' at ' + '%.2f' % interest + '%'
confirm_years = ' for ' + '%d' % years + ' years'
print confirm_principal + confirm_interest + confirm_years
```

The `+` is the concatenation of strings.
2. Confirm the User Input
using format operators and string concatenations

The `%` operator is used to confirm user input:
'
  `%2.f` to show floats with two digits after .
'`
  `%d` to convert an integer into a string.

Strings are stored in `confirm_principal`, `confirm_interest`, and `confirm_years`.

```python
# confirm the user input
confirm_principal = 'Investing ' \
    + '$%.2f' % principal
confirm_interest = ' at ' \
    + '%.2f' % interest + '%'
confirm_years = ' for ' + '%d' % years + ' years'
print confirm_principal + confirm_interest + confirm_years
```

The `+` is the concatenation of strings.
2. Confirm the User Input

using format operators and string concatenations

The % operator is used to confirm user input:
’%2.f’ to show floats with two digits after .
’%d’ to convert an integer into a string.

Strings are stored in confirm_principal, confirm_interest, and confirm_years.

```
# confirm the user input
confirm_principal = 'Investing ' \
    + '$%.2f' % principal
confirm_interest = ' at ' \
    + '%.2f' % interest + '%'
confirm_years = ' for ' + '%d' % years + ' years'
print confirm_principal + confirm_interest \
    + confirm_years
```

The + is the concatenation of strings.
2. Confirm the User Input

using format operators and string concatenations

The `%` operator is used to confirm user input:
'%.2f' to show floats with two digits after .
'%.d' to convert an integer into a string.
Strings are stored in confirm_principal, confirm_interest, and confirm_years.

# confirm the user input
confirm_principal = 'Investing ' \
    + '$%.2f' % principal
confirm_interest = ' at ' \
    + '%.2f' % interest + '%'
confirm_years = ' for ' + '%d' % years + ' years'
print confirm_principal + confirm_interest \
    + confirm_years

The + is the concatenation of strings.
3. Implementing the Algorithm

choosing good names for the variables

Obvious names for the results of the program are `balance` and `yield`.

But `yield` is a *keyword*. A keyword is a name reserved by the language and cannot be used as a variable.

```python
# calculate balance and yield
balance \n   = principal * (1.0 + interest/100.0)**years
the_yield = balance - principal
```

Note that Python is case sensitive: `yield` ≠ `Yield`. 
3. Implementing the Algorithm

choosing good names for the variables

Obvious names for the results of the program are balance and yield.

But yield is a keyword. A keyword is a name reserved by the language and cannot be used as a variable.

```python
# calculate balance and yield
balance \n    = principal * (1.0 + interest/100.0)**years
the_yield = balance - principal
```

Note that Python is case sensitive: yield ≠ Yield.
3. Implementing the Algorithm

choosing good names for the variables

Obvious names for the results of the program are *balance* and *yield*.

But *yield* is *a keyword*. A keyword is a name reserved by the language and cannot be used as a variable.

```python
# calculate balance and yield
balance \n  = principal * (1.0 + interest/100.0)**years
the_yield = balance - principal
```

Note that Python is case sensitive: *yield* ≠ *Yield*. 
3. Implementing the Algorithm

choosing good names for the variables

Obvious names for the results of the program are \texttt{balance} and \texttt{yield}.

But \texttt{yield} is a \textit{keyword}. A keyword is a name reserved by the language and cannot be used as a variable.

\begin{verbatim}
# calculate balance and yield
balance \ 
    = principal * (1.0 + interest/100.0)**years
the_yield = balance - principal
\end{verbatim}

Note that Python is case sensitive: \texttt{yield} \neq \texttt{Yield}.
4. Show the Results

using proper formats

We use `show_balance` and `show_yield` as variables to hold the output strings.

```python
# show the results to screen
show_balance = '$%.2f' % balance + ' as balance'
show_yield = '$%.2f' % the_yield + ' as yield'
print 'gives ' + show_balance + ' and ' + show_yield + '.
```

The complete program is posted at the course web site.
4. Show the Results

using proper formats

We use `show_balance` and `show_yield` as variables to hold the output strings.

```python
# show the results to screen
show_balance = '$%.2f' % balance + ' as balance'
show_yield = '$%.2f' % the_yield + ' as yield'
print 'gives ' + show_balance + ' and ' + show_yield + '.
```

The complete program is posted at the course web site.
4. Show the Results
using proper formats

We use `show_balance` and `show_yield` as variables to hold the output strings.

```python
# show the results to screen
show_balance = '$%.2f' % balance + ' as balance'
show_yield = '$%.2f' % the_yield + ' as yield'
print 'gives ' + show_balance + ' and ' + show_yield + '.
```

The complete program is posted at the course web site.
Variables and Assignments

Developing Python Programs

1. Variables and Assignments
   assignment operators
types and arithmetic

2. Developing Python Programs
   computing the yield of an investment
   from specification to implementation

3. Multiprecision Arithmetic
   accuracy and precision
   arbitrary precision in Sage

4. Summary + Assignments
Multiprecision Arithmetic
accuracy and precision

We distinguish between

- **precision**: size of numbers used in a computation
- **accuracy**: number of correct digits in data

Interested in seeing the first 100 decimal places of $\pi$?

```python
sage: help(sage.rings.mpfr)
sage: R = RealField(330)
sage: R.pi()
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211707
```

Why 330 bits?

```python
sage: 2.0**(-330)
4.57194956512910e-100
```
Multiprecision Arithmetic
accuracy and precision

We distinguish between

- precision: size of numbers used in a computation
- accuracy: number of correct digits in data

Interested in seeing the first 100 decimal places of $\pi$?

sage: help(sage.rings.mpfr)

sage: R = RealField(330)
sage: R.pi()
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211707

Why 330 bits?

sage: 2.0**(-330)
4.57194956512910e-100
Multiprecision Arithmetic
accuracy and precision

We distinguish between

- precision: size of numbers used in a computation
- accuracy: number of correct digits in data

Interested in seeing the first 100 decimal places of $\pi$?

```
sage: help(sage.rings.mpfr)
sage: R = RealField(330)
sage: R.pi()
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211707
```

Why 330 bits?

```
sage: 2.0**(-330)
4.57194956512910e-100
```
Variables and Assignments
Developing Python Programs

1. Variables and Assignments
   assignment operators
types and arithmetic

2. Developing Python Programs
   computing the yield of an investment
   from specification to implementation

3. Multiprecision Arithmetic
   accuracy and precision
   arbitrary precision in Sage

4. Summary + Assignments
Arbitrary Precision in Sage

Extending the working precision with Sage:

```
sage: R = RealField(100)
sage: two = R('2')
sage: type(two)
<type 'sage.rings.real_mpfr.RealNumber'>
sage: sqrt(two)
1.4142135623730950488016887242
```

A precision of 100 bits:

```
sage: 2.0**(-100)
7.88860905221012e-31
```

⇒ about 30 decimal places
Arbitrary Precision in Sage

Extending the working precision with Sage:

```python
sage: R = RealField(100)
sage: two = R('2')
sage: type(two)
<type 'sage.rings.real_mpfr.RealNumber'>
sage: sqrt(two)
1.4142135623730950488016887242
```

A precision of 100 bits:

```python
sage: 2.0**(-100)
7.88860905221012e-31
```

⇒ about 30 decimal places
Summary + Assignments

In this lecture we covered

- more of chapter 5 *Computer Science. An Overview*;
- start of chapter 4 of *Python Power!* on data types.

Assignments:

1. Write a Python program to implement the algorithm used in grocery stores to dispense change in coins. For example, $0.68 is dispensed as 2 quarters, 1 dime, 1 nickel, and 3 pennies, a minimal number of coins.

2. The formula $f = \frac{9}{5}c + 32$ converts $c$ degrees of Celsius into $f$ degrees of Fahrenheit. Write a Python program that prompts the user for $c$ and prints $f$.

3. Modify the yield and balance program for loans, i.e.: what is the balance of a loan when making a number of payments when the interest rate is fixed?