Outline

MCS 260 Lecture 4
Introduction to Computer Science
Jan Verschelde, 20 January 2016

Variables and Assignments
Developing Python Programs

constants and variables to store results of calculations

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

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

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

```
>>> 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**:

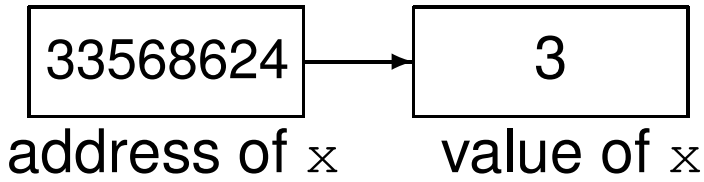
1. evaluate the right hand side of **=**
2. store the result of the evaluation in a register
3. compute the address of **x**
   and store it into the address register
4. copy the **3** from a register to the data register
5. put the value in data register at the memory location
   defined by the content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

Memory Locations

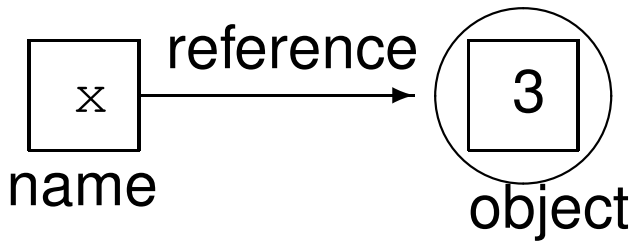
A variable has a value and an address.

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

The machine view:



In Python, the name `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:

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

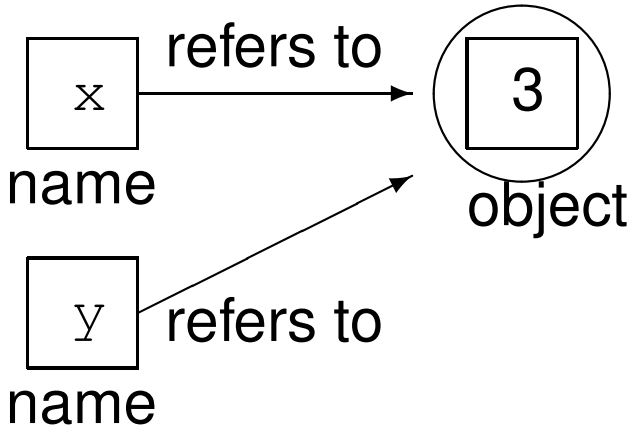
We see that both `x` and `y` refer to the same object.

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

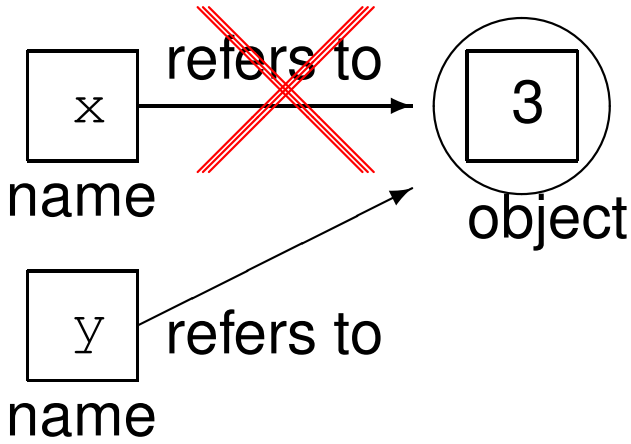
After deleting `x`, the object is still accessible via `y`.
Only after `del(y)` will memory space be released.

Deleting References

```
>>> x = 3; y = x
```




```
>>> del(x)
```



Assignment Operators

making updates to variables

Note that the assignment `x = 3` $\neq$ `x == 3`, the comparison!

Often assignments are updates to variables, therefore:

operator	example	description
`=`	`x = y`	afterwards: `x == y`
`+=`	`x += y`	equivalent to `x = x + y`
`-=`	`x -= y`	equivalent to `x = x - y`
`*=`	`x *= y`	equivalent to `x = x * y`
`/=`	`x /= y`	equivalent to `x = x / y`
`%=`	`x %= y`	equivalent to `x = x % y`

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

Multiple assignments `a = b = 1` useful as initializations.

Variables and Assignments
Developing Python Programs

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 + 2` $\neq$ `'1' + '2'`
- 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.

Integer Numbers

We distinguish between machine integers and long integers.
A Python programmer does *not* have to worry about this distinction.

- The computer uses 32 or 64 bits to represent integers.
  First bit is sign bit: $0 = +$, $1 = -$.
- The largest machine int is $+2^{31} - 1$ or $+2^{63} - 1$:

```
>>> 2**30 + (2**30 - 1)
2147483647
>>> 2**62 + (2**62 - 1)
9223372036854775807
```

- The smallest machine int is -2^{31} or -2^{63}:

```
>>> -2**31
-2147483648
>>> -2**63
-9223372036854775808
```

In Python, the use of long integers avoids overflow and underflow.

Floating-Point Numbers – machine precision

- 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 – floats to integers

```
>>> x = 3.7
>>> int(x)     # drops all digits after .
3
>>> round(x)
4
>>> type(round(x))
<type 'int'>
```

By default, `round` returns an integer number (in Python 2, this was a float) rounded with 0 digits after the decimal point.

```
>>> import math
>>> p = math.pi
>>> p
3.1415926535897931
>>> round(p,2)
3.14
```

Boolean

We can store the 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
   - prompting for user input
2. Developing Python Programs
   - computing the yield of an investment
   - from specification to implementation
   - the style guide for Python code
3. Multiprecision Arithmetic
   - accuracy and precision
   - arbitrary precision in Sage
4. Summary + Assignments

Getting User Input

unformatted or of specific type

Two ways to prompt the user for input:

1. `s = input('Give a number :  ')`
2. `n = int(input('Give a number :  '))`

In both cases, the user sees `Give a number :`

The difference is in the type of the object returned

1. `input()` returns always a string
2. `int(input())` converts the input string to an integer.

Why two ways?

1. the string returned by `input()` can be cast (converted, evaluated)
   into an integer or a float, depending on its purpose;
2. a direct conversion is good if we assume correct input.

Examples of Input

```
>>> n = input('give a number : ')
give a number : 5.6
>>> n
'5.6'
>>> type(n)
<type 'str'>
```

To compute with `n`, we must *convert* it to a float:

```
>>> f = float(n)
>>> f
5.6
```

input() determines the type of the given object:

```
>>> n = input('give a number : ')
give a number : 5.6
>>> type(n)
<type 'float'>
```

evaluation of Python literal structures

```
>>> from ast import literal_eval
>>> sn = input('give a number : ')
give a number : 3
>>> n1 = literal_eval(sn)
>>> n1
3
>>> type(n1)
<class 'int'>
>>> sf = input('give a number : ')
give a number : 3.4
>>> n2 = literal_eval(sf)
>>> n2
3.4
>>> type(n2)
<class 'float'>
```

With `literal_eval` Python evaluates a string into the proper type.

Guido van Rossum

author of the Python programming language

+ Python originated as “hobby” programming project during the Christmas break of 1989.
+ Emphasis on readability, using identation for statement grouping. Python is Free and Open Source.
+ Continues to oversee the Python development process. Python is one of the most popular languages.

personal home page at `www.python.org/~guido`.

Variables and Assignments
Developing Python Programs

1. Variables and Assignments
   - assignment operators
   - types and arithmetic
   - prompting for user input
2. Developing Python Programs
   - computing the yield of an investment
   - from specification to implementation
   - the style guide for Python code
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.

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
   - prompting for user input
2. Developing Python Programs
   - computing the yield of an investment
   - from specification to implementation
   - the style guide for Python code
3. Multiprecision Arithmetic
   - accuracy and precision
   - arbitrary precision in Sage
4. Summary + Assignments

Specification $\longrightarrow$ Implementation

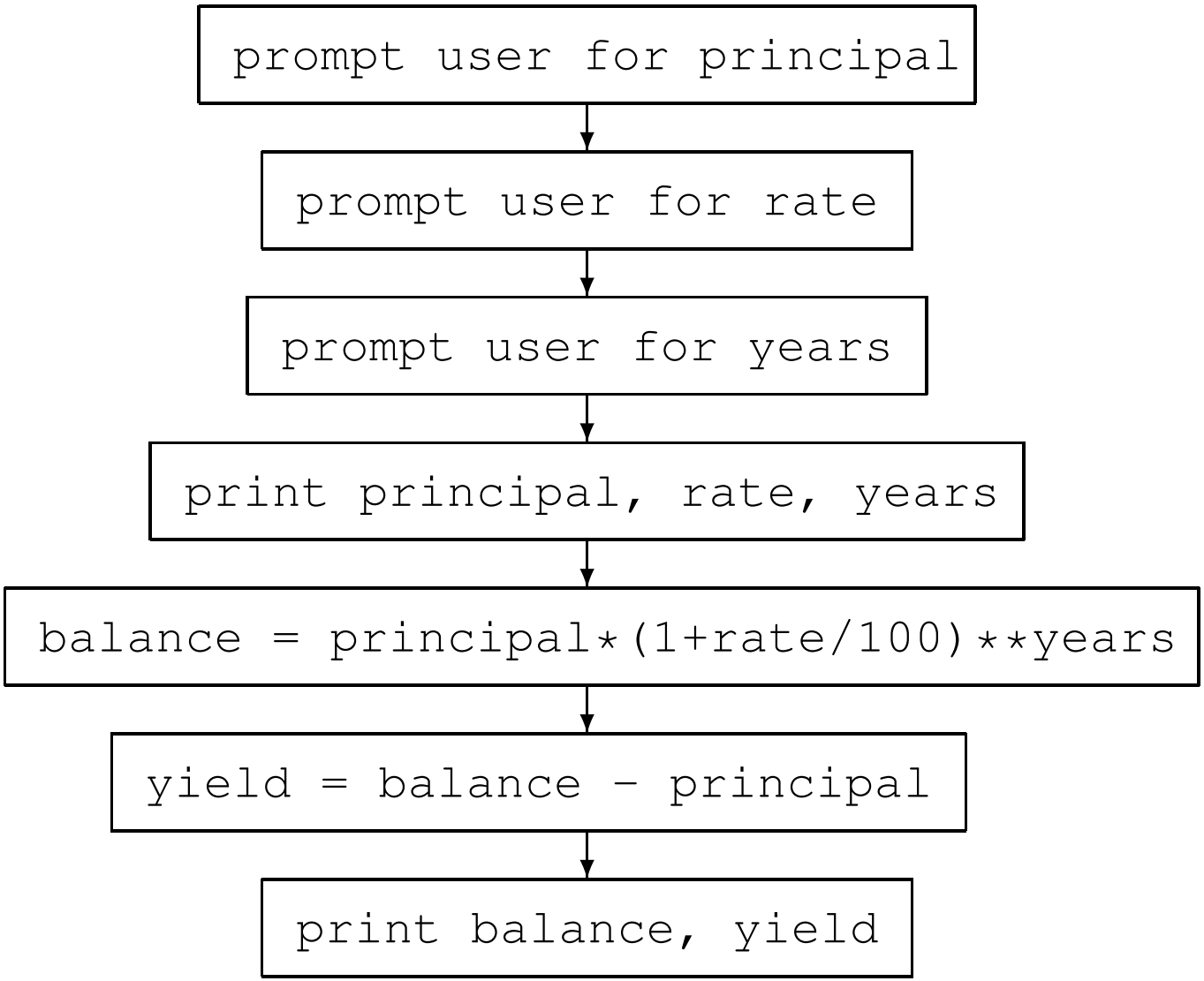
a top down incremental approach to programming

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

$B = P(1 + r/100)^n$



0. Structure of the Program

We write the program specification and pseudocode for the outline:

```
# L-4 MCS 260 : 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.
Type 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.

1. welcome message, read input and convert
2. confirm the user input
3. calculate balance and yield
4. show the results to screen
"""
```

1. Read Input from User (assuming all input is correct)

All output of `input` is assigned to `DATA`.
We calculate with `PRINCIPAL`, `INTEREST` and `YEARS`.

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

2. Confirm the User Input (adding formatted strings)

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 `INVESTING`, `RATE`, and `DURATION`.

```
# confirm the user input using string concatenation
INVESTING = 'Investing ' + '$%.2f' % PRINCIPAL
RATE = ' at ' + '%.2f' % INTEREST + '%'
DURATION = ' for ' + '%d' % YEARS + ' years'
print(INVESTING + RATE + DURATION)
```

The `+` is the concatenation of strings.

3. Implementing the Algorithm (naming variables)

Obvious names for the results are `BALANCE` and `YIELD`.

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

```
# calculate balance and yield
BALANCE = PRINCIPAL * (1.0 + INTEREST/100.0)**YEARS
YIELD = BALANCE - PRINCIPAL
```

Note that Python is case sensitive: yield $\neq$ Yield.

4. Show the Results (using proper formats)

We use `ENDBALANCE` and `SHOWYIELD` to store the output strings.

```
# print the outcome, again using string concatenation
ENDBALANCE = '$%.2f' % BALANCE + ' as balance'
SHOWYIELD = '$%.2f' % YIELD + ' as yield'
print('gives ' + ENDBALANCE + ' and ' + SHOWYIELD + '.')
```

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
   - prompting for user input
2. Developing Python Programs
   - computing the yield of an investment
   - from specification to implementation
   - the style guide for Python code
3. Multiprecision Arithmetic
   - accuracy and precision
   - arbitrary precision in Sage
4. Summary + Assignments

the style guide for Python code

Sloppily written code is hard to debug and maintain.
PEP = Python Enhancement Proposal

- PEP 8 – Style Guide for Python code
  is available at
  `http://legacy.python.org/dev/peps/pep-0008/`
- The style guide gives coding conventions for Python, e.g.: on
  - indentation of 4 spaces,
  - comments and documentation strings,
  - naming conventions of variables.
    Avoid the little l, because it looks too much like 1.
- `www.pylint.org` provides free software to check your code against the Python coding conventions.

Variables and Assignments
Developing Python Programs

1. Variables and Assignments
   - assignment operators
   - types and arithmetic
   - prompting for user input
2. Developing Python Programs
   - computing the yield of an investment
   - from specification to implementation
   - the style guide for Python code
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 π?

```
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
   - prompting for user input
2. Developing Python Programs
   - computing the yield of an investment
   - from specification to implementation
   - the style guide for Python code
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
```

$\Rightarrow$ about 30 decimal places

Summary

In this lecture we introduced the assignment operation.

The outcome of operations depends on the type of variables.

Python programmers do not have to worry about the shortcomings of machine integers, but floating-point numbers introduce errors.

We developed a first complete Python script.

Background literature for this lecture:

- more of chapter 5 *Computer Science. An Overview*;
- §1.5.2 of *Python Programming in Context*.

Assignments

1. Can you explain why `1.1 + 0.1` shows `1.2000000000000002`. Why is the outcome of `1.1 + 0.1` not exact?
2. Explain why the string `'%.40e' % .5` is exactly the same as `0.5` while `'%.40e' % .1` differs from the exact `0.1`.
3. Write a Python program to convert a duration expressed in seconds in an hour, minutes, and seconds format. For example: 35781 seconds equals 9 hours, 56 minutes, and 21 seconds.
4. 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.
5. To compute the present value of a future sum we can invert the formula $B = P(1 + r/100)^n$, as $P = B(1 + r/100)^{-n}$.
   If we want the balance to be B after n years invested at a rate r, then how large should our investment P be?
   Modify the `yieldbal.py` script to answer this question.