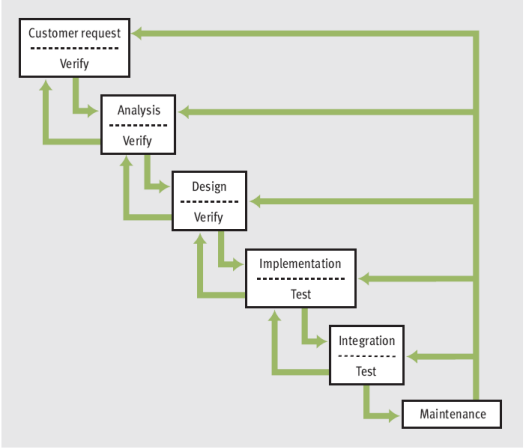
The Software Development Process

- **Software development:** process of planning and organizing a program
  - Several approaches; one is the **waterfall model**
- Modern software development is usually **incremental** and **iterative**
  - Analysis and design may produce a **prototype** of a system for coding, and then back up to earlier phases to fill in more details after some testing

The Software Development Process (continued)

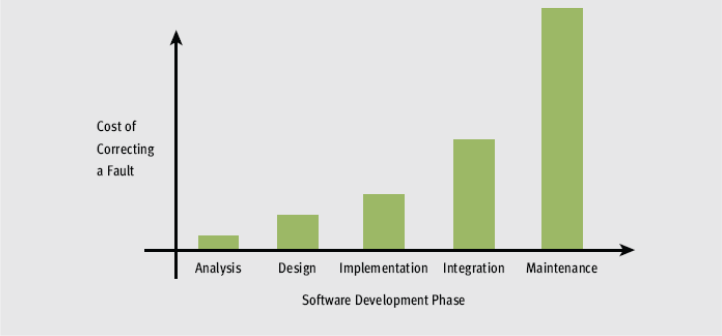


[FIGURE 2.1] The waterfall model of the software development process

The Software Development Process (continued)

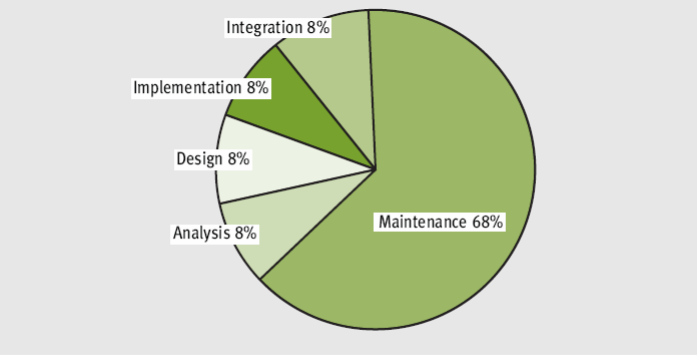
- Programs rarely work as hoped the first time they are run
  - Must perform extensive and careful testing
  - The cost of developing software is not spread equally over the phases

The Software Development Process (continued)



[FIGURE 2.2] Relative costs of repairing mistakes that are found in different phases

The Software Development Process (continued)



[FIGURE 2.3] Percentage of total cost incurred in each phase of the development process

Case Study: Income Tax Calculator

- Each year nearly everyone faces the unpleasant task of computing his or her income tax return
- If only it could be done as easily as suggested in this case study
- We begin with the **request**:
  - a program that computes a person's income tax

Case Study: Analysis

- All taxpayers are charged a flat tax rate of 20%
- Taxpayers are allowed $10,000 standard deduction
- For each dependent, taxpayer is allowed additional $3000 deduction
- Gross income must be entered to nearest penny
- Income tax is expressed as decimal number

```
Enter the gross income: 150000.00
Enter the number of dependents: 3
The income tax is $26200.00
```

[FIGURE 2.4] The user interface for the income tax calculator

Case Study: Design

- Algorithms are often written in a somewhat stylized version of English called **pseudocode**
- Pseudocode for our income tax program:
  - Input the gross income and number of dependents
  - Compute the taxable income using the formula
  - Taxable income = gross income - 10000 - (3000 * number of dependents)
  - Compute the income tax using the formula
  - Tax = taxable income * 0.20
  - Print the tax

Case Study: Implementation (Coding)

```
"""
Program: taxform.py
Author: Ken Lambert

Compute a person's income tax.

1. Significant constants
       tax rate
       standard deduction
       deduction per dependent
2. The inputs are
       gross income
       number of dependents
3. Computations:
       taxable income = gross income - the standard deduction -
                        a deduction for each dependent
       income tax = is a fixed percentage of the taxable income
4. The outputs are
       the income tax
"""

# Initialize the constants
TAX_RATE = 0.20
STANDARD_DEDUCTION = 10000.0
DEPENDENT_DEDUCTION = 3000.0
```

continued

Case Study: Implementation (Coding) (continued)

```
# Request the inputs
grossIncome = float(input("Enter the gross income: "))
numDependents = int(input("Enter the number of dependents: "))

# Compute the income tax
taxableIncome = grossIncome - STANDARD_DEDUCTION - \
                DEPENDENT_DEDUCTION * numDependents
incomeTax = taxableIncome * TAX_RATE

# Display the income tax
print("The income tax is $" + str(incomeTax))
```


Case Study: Testing

- Even if there are no syntax errors, the program could still have a **logic error** or a **design error**
- May use a **test suite** to test if program is **correct**

Case Study: Testing (continued)

NUMBER OF DEPENDENTS	GROSS INCOME	EXPECTED TAX
0	10000	0
1	10000	–600
2	10000	–1200
0	20000	2000
1	20000	1400
2	20000	800

[TABLE 2.1] The test suite for the tax calculator program

Strings, Assignment, and Comments

- Text processing is by far the most common application of computing
  - E-mail, text messaging, Web pages, and word processing all rely on and manipulate data consisting of strings of characters

Data Types

- A **data type** consists of a set of values and a set of operations that can be performed on those values
- A **literal** is the way a value of a data type looks to a programmer
- `int` and `float` are **numeric data types**

Data Types (continued)

TYPE OF DATA	PYTHON TYPE NAME	EXAMPLE LITERALS
Integers	`int`	`-1, 0, 1, 2`
Real numbers	`float`	`-0.55, .3333, 3.14, 6.0`
Character strings	`str`	`"Hi", "", 'A', '66'`

[TABLE 2.2] Literals for some Python data types

String Literals

- In Python, a string literal is a sequence of characters enclosed in single or double quotation marks
- `''` and `""` represent the **empty string**
- Use `'''` and `"""` for multi-line paragraphs

```
>>> "I'm using a single quote in this string!"
"I'm using a single quote in this string!"
>>> print("I'm using a single quote in this string!")
I'm using a single quote in this string!
>>>
>>> print("""This very long sentence extends all the way to
the next line.""")
This very long sentence extends all the way to
the next line.
```

```
>>> """This very long sentence extends all the way to
the next line. """
'This very long sentence extends all the way to\nthe next line.'
>>>
```


Escape Sequences

- The newline character **\n** is called an **escape sequence**

ESCAPE SEQUENCE	MEANING
`\b`	Backspace
`\n`	Newline
`\t`	Horizontal tab
`\\`	The \ character
`\'`	Single quotation mark
`\"`	Double quotation mark

[TABLE 2.3] Some escape sequences in Python

String Concatenation

- You can join two or more strings to form a new string using the concatenation operator **+**
- The ***** operator allows you to build a string by repeating another string a given number of times

```
>>> " " * 10 + "Python"
'          Python'
>>>
```


Variables and the Assignment Statement

- A **variable** associates a name with a value
  - Makes it easy to remember and use later in program
- Variable naming rules:
  - Reserved words cannot be used as variable names
    - Examples: **`if`**, **`def`**, and **`import`**
  - Name must begin with a letter or _
  - Name can contain any number of letters, digits, or _
  - Names are case sensitive
    - Example: **`WEIGHT`** is different from **`weight`**
  - Tip: use “camel casing” (Example: **`interestRate`**)

Variables and the Assignment Statement (continued)

- Programmers use all uppercase letters for **symbolic constants**
  - Examples: **`TAX_RATE`** and **`STANDARD_DEDUCTION`**
- Variables receive initial values and can be reset to new values with an **assignment statement**

  <variable name> = <expression>
  - Subsequent uses of the variable name in expressions are known as **variable references**

```
>>> firstName = "Ken"
>>> secondName = "Lambert"
>>> fullName = firstName + " " + secondName
>>> fullName
'Ken Lambert'
>>>
```


Program Comments and Docstrings

- **Docstring** example:

```
"""
Program: circle.py
Author: Ken Lambert
Last date modified: 2/10/11

The purpose of this program is to compute the area of a circle.
The input is an integer or floating-point number representing the
radius of the circle. The output is a floating-point number
labeled the area of the circle.
"""
```

- **End-of-line comment** example:

```
>>> RATE = 0.85   # Conversion rate for Canadian to US dollars
```

Numeric Data Types and Character Sets

- The first applications of computers were to crunch numbers
- The use of numbers in many applications is still very important

Integers

- In real life, the range of **integers** is infinite
- A computer's memory places a limit on magnitude of the largest positive and negative integers
  - Python's **`int`** typical range: -2^{31} to $2^{31} - 1$
- Integer literals are written without commas

Floating-Point Numbers

- Python uses **floating-point** numbers to represent real numbers
- Python's `float` typical range: -10^{308} to 10^{308} and
- Typical precision: 16 digits

Floating-Point Numbers (continued)

DECIMAL NOTATION	SCIENTIFIC NOTATION	MEANING
3.78	3.78e0	3.78×10^{0}
37.8	3.78e1	3.78×10^{1}
3780.0	3.78e3	3.78×10^{3}
0.378	3.78e-1	3.78×10^{-1}
0.00378	3.78e-3	3.78×10^{-3}

[TABLE 2.4] Decimal and scientific notations for floating-point numbers

Character Sets

	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DCI	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	“	#	$	%	&	`
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	‘	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

[TABLE 2.5] The original ASCII character set

Character Sets (continued)

- In Python, character literals look just like string literals and are of the string type
  - They belong to several different **character sets**, among them the **ASCII set** and the **Unicode set**
- ASCII character set maps to set of integers
- **`ord`** and **`chr`** convert characters to and from ASCII

```
>>> ord('a')
97
>>> ord('A')
65
>>> chr(65)
'A'
>>> chr(66)
'B'
>>>
```

Expressions

- A literal evaluates to itself
- A variable reference evaluates to the variable's current value
- **Expressions** provide easy way to perform operations on data values to produce other values
- When entered at Python shell prompt:
  - an expression's operands are evaluated
  - its operator is then applied to these values to compute the value of the expression

Arithmetic Expressions

- An **arithmetic expression** consists of operands and operators combined in a manner that is already familiar to you from learning algebra

OPERATOR	MEANING	SYNTAX
-	Negation	-a
**	Exponentiation	a ** b
*	Multiplication	a * b
/	Division	a / b
//	Quotient	a // b
%	Remainder or modulus	a % b
+	Addition	a + b
-	Subtraction	a - b

[TABLE 2.6] Arithmetic operators

Arithmetic Expressions (continued)

- **Precedence rules**:
  - ** has the highest precedence and is evaluated first
  - Unary negation is evaluated next
  - *, /, and % are evaluated before + and -
  - + and - are evaluated before =
  - With two exceptions, operations of equal precedence are **left associative**, so they are evaluated from left to right
    - ** and = are **right associative**
  - You can use () to change the order of evaluation

Arithmetic Expressions (continued)

EXPRESSION	EVALUATION	VALUE
`5 + 3 * 2`	`5 + 6`	`11`
`(5 + 3) * 2`	`8 * 2`	`16`
`6 % 2`	`0`	`0`
`2 * 3 ** 2`	`2 * 9`	`18`
`-3 ** 2`	`-(3 ** 2)`	`-9`
`-(3) ** 2`	`9`	`9`
`2 ** 3 ** 2`	`2 ** 9`	`512`
`(2 ** 3) ** 2`	`8 ** 2`	`64`
`45 / 0`	`Error: cannot divide by 0`	
`45 % 0`	`Error: cannot divide by 0`	

[TABLE 2.7] Some arithmetic expressions and their values

- 45%0 is a **semantic error**

Arithmetic Expressions (continued)

- When both operands of an expression are of the same numeric type, the resulting value is also of that type
- When each operand is of a different type, the resulting value is of the more general type
  - Example: `3 / 4.0` is `.75`
- For multi-line expressions, use a \

```
>>> 3 + 4 * \
2 ** 5
131
>>>
```


Mixed-Mode Arithmetic and Type Conversions

- **Mixed-mode arithmetic** involves integers and floating-point numbers:

```
>>> 3.14 * 3 ** 2
28.26
```

- **Remember**—Python has different operators for quotient and exact division:

`3 // 2 * 5.0` yields `1 * 5.0`, which yields `5.0`

`3 / 2 * 5` yields `1.5 * 5`, which yields `7.5`

Tip:
  - Use exact division
  - Use a **type conversion function** with variables

Mixed-Mode Arithmetic and Type Conversions (continued)

CONVERSION FUNCTION	EXAMPLE USE	VALUE RETURNED
int(<a number or a string>)	int(3.77)	3
	int("33")	33
float(<a number or a string>)	float(22)	22.0
str(<any value>)	str(99)	'99'

[TABLE 2.8] Type conversion functions

Mixed-Mode Arithmetic and Type Conversions (continued)

- Note that the **`int`** function converts a **`float`** to an **`int`** by truncation, not by rounding

```
>>> int(6.75)
6
>>> round(6.75)
7
```

Mixed-Mode Arithmetic and Type Conversions (continued)

- Type conversion also occurs in the construction of strings from numbers and other strings

```
>>> profit = 1000.55
>>> print('$' + profit)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'float' objects
```

- Solution: use **`str`** function

```
>>> print('$' + str(profit))
$1000.55
```

- Python is a strongly **typed** programming language

Using Functions and Modules

- Python includes many useful functions, which are organized in libraries of code called **modules**

Calling Functions: Arguments and Return Values

- A **function** is chunk of code that can be called by name to perform a task
- Functions often require **arguments** or **parameters**
  - Arguments may be **optional** or **required**
- When function completes its task, it may **return a value** back to the part of the program that called it

```
>>> help(round)

Help on built-in function round in module builtin:

round(...)
    round(number[, ndigits]) -> floating point number

    Round a number to a given precision in decimal digits (default 0 digits).
    This returns an int when called with one argument, otherwise the same type as
    number. ndigits may be negative.
```


The `math` Module

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atanh', 'ceil','copysign', 'cos', 'cosh', 'degrees', 'e',
'exp', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'hypot',
'isinf', 'isnan', 'ldexp', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

- To use a resource from a module, you write the name of a module as a qualifier, followed by a dot (**.**) and the name of the resource
  - Example: **`math.pi`**

```
>>> math.pi
3.1415926535897931
>>> math.sqrt(2)
1.4142135623730951
```

The `math` Module (continued)

- You can avoid the use of the qualifier with each reference by importing the individual resources

```
>>> from math import pi, sqrt
>>> print(pi, sqrt(2))
3.14159265359 1.41421356237
>>>
```

- You may import all of a module's resources to use without the qualifier
  - Example: **`from math import *`**

Program Format and Structure

- Start with comment with author's name, purpose of program, and other relevant information
  - In a docstring
- Then, include statements that:
  - Import any modules needed by program
  - Initialize important variables, suitably commented
  - Prompt the user for input data and save the input data in variables
  - Process the inputs to produce the results
  - Display the results

Running a Script from a Terminal Command Prompt (continued)

- Python installations enable you to launch Python scripts by double-clicking the files from the OS's file browser
  - May require **`.py`** file type to be set
  - Fly-by-window problem: Window will close automatically
    - Solution: Add an input statement at end of script that pauses until the user presses the enter or return key

```
input("Please press enter or return to quit the program. ")
```


Summary

- Waterfall model describes software development process in terms of several phases
- Literals are data values that can appear in program
- The string data type is used to represent text for input and output
- Escape characters begin with backslash and represent special characters such as delete key
- A docstring is string enclosed by triple quotation marks and provides program documentation

Summary (continued)

- Comments are pieces of code not evaluated by the interpreter but can be read by programmers to obtain information about a program
- Variables are names that refer to values
- Some data types: **`int`** and **`float`**
- Arithmetic operators are used to form arithmetic expressions
  - Operators are ranked in precedence
- Mixed-mode operations involve operands of different numeric data types

Summary (continued)

- A function call consists of a function's name and its arguments or parameters
  - May return a result value to the caller
- Python is a strongly typed language
- A module is a set of resources
  - Can be imported
- A semantic error occurs when the computer cannot perform the requested operation
- A logic error produces incorrect results


EAST CENTRAL UNIVERSITY
1909