SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 120: Introduction to Computing Science and Programming 1

Procedural programming in Python

python™

Reminders

One-Stop Access To Course Information

- **Course website: One-stop access** to all course information.

  http://www2.cs.sfu.ca/CourseCentral/120/liaqata/WebSite/index.html

- Course Outline
- Exam Schedule
- Python Info
- CourSys/Canvas link
- Learning Outcomes
- Office Hours
- Textbook links
- and more…
- Grading Scheme
- Lab/Tutorial Info
- Assignments

- **Canvas**: Discussions forum - https://canvas.sfu.ca/courses/39187
- **CourSys**: Assignments submission, grades - www.coursys.sfu.ca

How to Learn in This Course?

A **Attend** Lectures & Labs

R **Read** / review Textbook/Slides/Notes

R **Reflect** and ask Questions

O **Organize** – your learning activities on weekly basis,

and finally…

W **Write** Code, Write Code, and Write Code.

Deliverables

1. Deliverables are due by the given date and time.
2. For the course, we are using IDLE to write and run our Python code.
3. You can use the CSIL lab computers outside your lab hours.
4. Plan ahead your assignments and other deliverables. Computer crash, network problems etc. are not acceptable excuses for delays in deliverables.
5. You may use online Python interpreters for running and testing your codes, such as:

   https://repl.it/languages/Python3

Labs

1. Each lab has an assigned TA.
2. Attend your assigned lab and show your work to your TA for the participation marks.
3. Class enrolments and lab swaps are closed now.

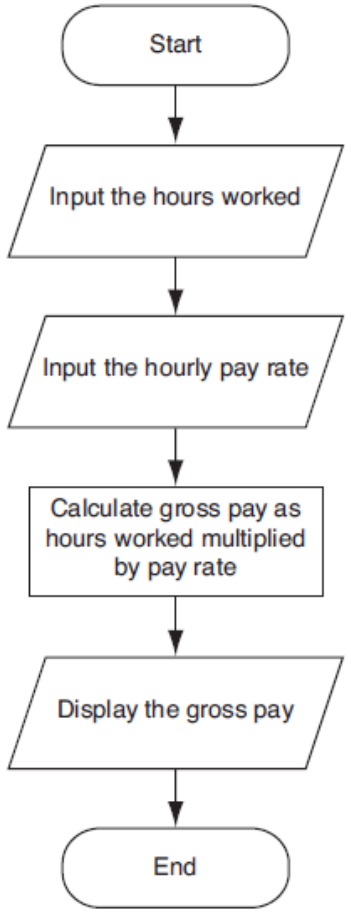
Course Topics

1. General introduction
2. Algorithms, flow charts and pseudocode
3. Procedural programming in Python
4. **Data types and control structures**
5. Fundamental algorithms
6. Binary encodings
7. Basics of computability and complexity
8. Basics of Recursion
9. Subject to time availability:
   - Basics of Data File management

Today's Topics

1. Coding Practice
2. Input / Output
   - Functions
3. Types
   - Type Conversion
4. Order of operations
5. String operations

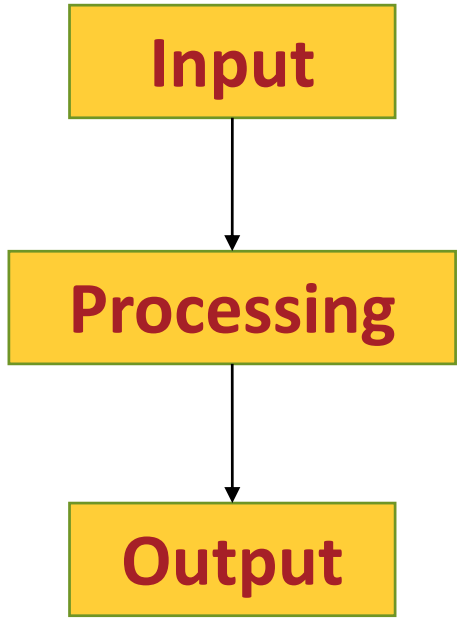
Coding Practice: Write Code From Flowchart



Liaqat Ali, Summer 2018.

Computer – A Data Processing Machine

- Computers process data.
- Data processing is typically a 3-step process.
  1. **Input:** Receive input.
     - Input: any data that the program receives while it is running
  2. **Process**: Perform some process on the input.
     - Example: mathematical calculation
  3. **Output**: Produce output



Getting Input From Users

- `Input()`: We use input function to get input from users.
- Input function lets you ask a user for some **text** input.
- When you **call** this function, your program stops and waits for the user to key in the data.
- When a user enters data, the input function **returns** the data as a **string**.
- **Format**: variable = `input(`prompt`)`
  - prompt is usually a **string** to instruct to user to enter a value.
  - The input function does not automatically display a space after the prompt.
  - Example: midterm_marks = `input(`“Enter midterm marks: ”`)`

Reading Numbers with the *input* Function

- `input()`: function always returns a string.
- Python provides us more built-in functions to convert the string data type into number data types.
  1. `int(`*string_data_argument*`)` converts *string_data type* to an int type.
  2. `float(`*string_data_argument*`)` converts *string_data type* to an float type.
- Nested function call: We can nest one function inside another function.
  - **Format**: ***function1(function2(argument))***
    - A value returned by **function2** will be passed to **function1**.
    - midterm_marks = **float(input(“Enter marks: “))**
  - Type conversion only works if *string_data_argument* is a valid numeric value, otherwise, program with throw an **error message**, or **exception**.

Functions

- **Function**: Function is a piece (or a block) of reusable code written to perform a single, related task.
  1. We can write functions inside our programs.
     - We haven't done that yet. But, we will do it later in coming weeks.
  2. Python provides us with some pre-written functions to use in our programs. For example, `input()` is a pre-written Python function.
  3. We may **send** some data to a function, inside parenthesis, to process it.
     1. We call this data as **arguments**.
  4. A function may **return** us back some result.
  5. We also call these pre-written Python function as **built-in** functions.
  6. We can name functions like variable.
  7. How to identify a function? Functions names are followed by a pairs of parenthesis **()**

Displaying Output From Your Program

- **`print()`**: Use **output** function to display a **line of output** from a program.
  - Newline character (' \n' } at end of printed data. (Moves cursor to next line.)
- **Format**: `print(value1, value2, . . . , sep = ' ', end = 'delimiter')`

```
>>> print(67)
>>> 67
```

- **`print()`** accepts multiple items as arguments.

```
>>> print(total, gpa)
>>> ____________________
```

- **`print()`** uses space as item separator by default.

```
>>> print(total, gpa, sep = ',')
>>> ____________________
```

Displaying Output From Your Program

- Special argument `end='delimiter'` causes `print` to place *delimiter* at end of data instead of newline character.

```
>>> print(total, gpa, end = '.')
>>> _____
```

- We can also use following special characters inside string literals.
  - Preceded by backslash (\): **newline (\n), horizontal tab (\t)**
  - They are treated as commands embedded in strings.

```
>>> print(total, gpa, sep = '\t', end = '.\n\n')
>>> _____
```

- + operator between two strings performs string concatenation
  - Useful for breaking up a long string literal into more than one literals

```
>>> print('This will \t' + 'be joined together. \n')
```

Displaying Output From Your Program

- Using round() to format output.

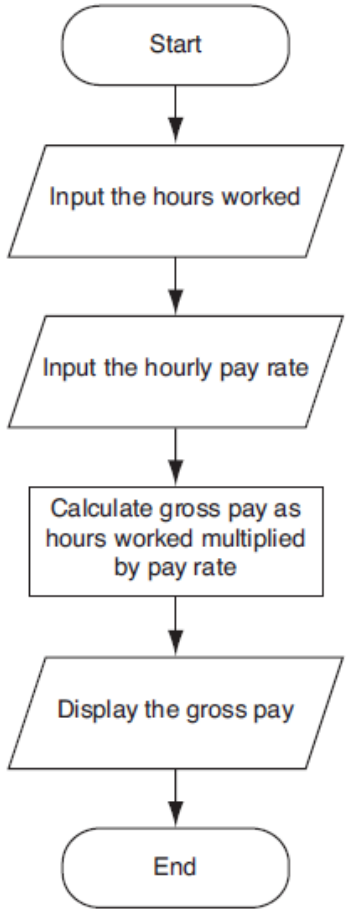
```
>>> exchangeRate = 3.476524
>>> round(exchangeRate, 2)
>>> _____________________
```

- We can also use % as place holder for values with a format code and number.

```
>>> item = 'Chair'
>>> cost = 200
>>> print(item, cost)

_____________________________
Price of a Chair is 200.
>>> print('Price of a %s is %d.' %(item, cost))
>>> print('Price of a %10s is %10d.' %(item, cost))
```

Identify Variables, Functions, Arguments, Operator



Liaqat Ali, Summer 2018.

```
hours_worked = input("Enter hours worked: ")
hourly_rate  = input("Enter hourly rate: ")
gross_pay = float(hours_worked) * float(hourly_rate)
print(gross_pay)
```

Variables

Strings literal used as argument

functions

Variable used as argument

Operator

Values: Types

- **Values**: A value is one of the basic things (data) a program works with, like a letter or a number.
- A value can be of different type (or category or class):
  - **Number**
    - **Integer (int)**
    - **Float-point number (float)**
  - **String (str)**
- **type():** The type() function can be used to check type of a value in the shell.

```
>>> type(2)
```

```
>>> type(42.0)
```

```
>>> type('Hello, World!')
```

Formal and Natural Languages

- **Natural Languages**: The languages people speak, such as English or Spanish.
- **Formal Languages**: The languages that are designed by people for specific applications. For example, programming languages, such as Python or C++.
  - These languages are designed to express computations.
- Formal languages tend to have strict structure of statements, called **syntax**.

Read section 1.6, of textbook Think Python for details.

Debugging

- Programmers make mistakes. Programming errors are called **bugs**
- **Debugging**: Debugging is a process finding and fixing errors in the code.
- Be prepared:
  - Programming, and especially debugging, sometimes brings out strong emotions.
  - If you are struggling with a difficult bug, you might feel angry, despondent, or embarrassed.

Read section 1.7, of textbook Think Python for details.

Comments

- As programs get **bigger** and **complicated**, they get more difficult to read.
- **Formal languages** are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.
- For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing.
- These notes are called **comments**, and they start with the # symbol:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

- This comment contains useful information that is not in the code:

```
v = 5      # velocity in meters/second.
```

Practice: Add Comments To The Following Code

```
hours_worked = 0
hourly_rate  = 0
gross_pay =  0

hours_worked = input("Enter hours worked: ")
hourly_rate  = input("Enter hourly rate: ")
gross_pay = float(hours_worked) * float(hourly_rate)
print(gross_pay)
```

Never submit your code without comments.

No comments means your program is not yet ready or submission.

Comment Your Programs: Sample

```
# Gross_Pay.py
#
# In-Class Practice
# Gross Pay Computation
#
# Liaqat Ali
# May 2018

# Set up pay variables
hours_worked = 0
hourly_rate  = 0
gross_pay =  0

# Ask the customer for the input
# Get the number of hours worked by the employee.
hours_worked = input("Enter hours worked: ")

# Get the per hour rate for the employee.
hourly_rate  = input("Enter hourly rate: ")

# Compute the gross pay
gross_pay = float(hours_worked) * float(hourly_rate)

# Display the gross pay earned by the employee
print(gross_pay)
```

Class Participation

Class participation. (Due tonight): Add comments to the attached program (as shown on lecture slide 23: Comment Your Programs: Sample) and **post it** on the **Canvas Discussions forum** by **tonight 11:59pm**.

(Note: You would comment all your future programs in the same way. Try to create your personalized comments template, and save it on your computer.)

Order of Operations

- When an expression contains more than one operator, the order of evaluation **depends on the order of operations**.
- For mathematical operators, Python follows mathematical convention. The acronym PEMDAS is a useful way to remember the rules:
  - **Parentheses** have the highest precedence and can be used to force an expression to evaluate in the order you want.
  - Expressions in parentheses are evaluated first, **2 * (3-1)** is **4**, and **(1+1)**(5-2)** is **8**.
  - You can also use parentheses to make an expression easier to read, even if it doesn't change the result. As in: (minute * 100) / 60.

Order of Operations

- **Exponentiation** has the next highest precedence, so: **1 + 2**3** is **9**, **not 27,** and **2*3**2** is **18**, not 36.
- Multiplication and Division have higher precedence than **Addition** and **Subtraction**. So:
  2*3-1 is **5**, not 4, and **6+4/2** is 8, not 5.
- Operators with the **same precedence** are evaluated from **left to right** (except exponentiation).
- So in the expression **degrees / 2 * pi**, the division happens first and the result is multiplied by pi. To divide by 2pi, you can use parentheses **degrees / (2 * pi)** or write **degrees / 2 / pi**.

String Operations

- We can't perform mathematical operations on strings. The following is illegal:
  '2'-'1' 'eggs'/'easy' 'third'*'a charm'
- **But there are two exceptions, + and *.**
  - **The + operator performs string concatenation, which means it joins the strings by linking them end-to-end. For example:**

```
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
Throatwarbler
```

  The * operator performs repetition on strings. For example, **'Spam'*3** is **'SpamSpamSpam'**. If one of the values is a string, the other has to be an integer.

Questions?