Introduction to Computer Programming
-
Dealing with Data

Summer 2022

Values

Values = Data

- for now, a **value** is a number or a string (we'll see more **types** later)
- it can be stored into a **variable**
- it can be part of an **expression** (a combination of values, operators, functions, . . . that can be evaluated)
- but it can't be *evaluated* on its own (`2 + 3` is not a value, because it can be evaluated further)

Some examples of values:

- `-123456` is an *integer literal*
- `'a string is a value'` is a *string literal*

Data types

Classification of values

Values can be classified by **Data Type**

Today, we will go over the following 4 types:

- `str` (string)
- `int` (integer)
- `float` (floating point)
- `complex` (complex number)

The last 3 types are **numeric types**.

Strings

What is a string?

A **string** is a **sequence of characters**.

- any characters
- including spaces and punctuation
- must **start and end** with quotes! (*string delimiters*)

String delimiters

We have multiple choices for **string delimiters**:

- single quotes: `'`
- double quotes: `"`
- triple single quotes: `'''`
- triple double quotes: `"""`

Triple single/double quotes can be used for **multiline strings**

Strings

Examples

```
'Single quoted string'
```

```
"Double quoted string"
```

```
'''First Line
Second line'''
```

```
"""1
2
3"""
```

Strings

Unbalanced quotes

When you define a string, you need to be careful with the **starting and ending quotes**.
For example, an extra quote in the string will lead to a **Syntax Error** since it is ambiguous which quote is the ending one.

Example:

```
>>> "Try "additional quotes"
SyntaxError:  invalid syntax
```

Strings

Workaround 1: Mixing quotes

If your string contains a particular quote, you have to choose another quote for string delimiters.

Example:

- `'Try "additional quotes'`
- `"triple single quotes '''"`

Workaround 2: escape sequence

Escape sequence can be used to obtain quote character in a string.
Escape sequence starts with a **backslash symbol** `\`

- `"Try \"additional quotes"`
- `'triple single quotes \'\'\''`

Strings

Escape Sequence

Escape Sequence	Meaning
`\\`	Backslash (\)
`\'`	Single quote (')
`\"`	Double quote (")
`\a`	ASCII Bell (BEL)
`\b`	ASCII Backspace (BS)
`\f`	ASCII Formfeed (FF)
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\v`	ASCII Vertical Tab (VT)

Numeric types

Numeric types

The following types are all closely related and most of the same operations can be applied:

- `int` (integer)
- `float` (floating point)
- `complex` (complex number)

Numeric types

`int` (integer)

- integer: whole number, can be negative
- no size limit (as much as your computer can handle)!

Examples

- `24`
- `-61`
- `16 * 88`
- `1337 ** 20`
- `2456 ** 10000`

Numeric types

`float` (floating point)

A **floating point number** represents a **decimal fraction**
It differs from *integer* with the presence of a **decimal point**

Example:

- `-5.5555` is a `float`
- `14.0` is a `float`
- `14` is an `int`

Numeric types

Scientific notation

You can input **float numbers** using **scientific notation** using `e` for decimal exponentiation
Example:

- `2e+3` is equal to $2 \cdot 10^3$
- `4e-5` is equal to $4 \cdot 10^{-5}$
- `123.456e100`

Numeric types

Comment about float accuracy

A float is also coded with **binary** digits.
It actually uses some **negative powers of 2**.

Question: Can we represent exactly `0.2` on a float?

Let's give it a try

$$
\begin{aligned}
0.2 &= 0.125 + \ldots \\
&= 0.125 + 0.075 \\
&= 0.125 + 0.0625 + \ldots \\
&= 0.125 + 0.0625 + 0.0125 \\
&= 0.125 + 0.0625 + 0.0078125 + \ldots \\
&= 0.125 + 0.0625 + 0.0078125 + 0.0046875 \\
&= \ldots
\end{aligned}
$$

Answer: It needs an infinite number of bits...
The number will be **truncated**.

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	...
0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	...

Numeric types

`complex` (complex numbers)

Python also supports **complex numbers** (numbers with a real and an imaginary part)
The imaginary part is indicated with character `j` (square root of -1)

Example:

- `1j`
- `2 - 3j`
- `2.4 + 3.6j`
- `1j * 1j`

Numeric types

Mixing numeric types

Python is able to mix numeric types during **arithmetic operations**. The operand with the "*narrower*" type will be widened to that of the other: `int` $\subset$ `float` $\subset$ `complex`

Examples:

- `2` is an `int`
- `3.0` is a `float`
- `1.1 + 5j` is a `complex`
- `2 + 3.0` is a `float`
- `2 * 3` is an `int`
- `2 * 3.0` is a `float`
- `2 * -3` is an `int`
- `2 ** 3` is an `int`
- `2 ** 3.0` is a `float`
- `2 ** -3` is a `float`
- `2 ** 3j` is a `complex`
- `1j * 1j` is a `complex`
- `2 / 3` is a `float`
- `6j / 3j` is a `complex`

Data types

How to know which type a value is?

We can use the *built-in function* `type`, it displays in the shell what type the value is.

Example:

- `type(5)`

```
<class 'int'>
```

- `type(2.0)`

```
<class 'float'>
```

- `type('hello')`

```
<class 'str'>
```

Data types

Conversion

Python provides several *builtin functions* that can convert **Data types** into other types.

The functions are `int()`, `float()`, `complex()` and `str()`.

`int()` function

`int()` can convert a string or a float into an int.

Examples:

```
>>> int("123")
123
```

```
>>> int(456.78)
456
```

Data types

`float()` function

`float()` can convert an int or a string into an float.

Examples:

```
>>> float("123.456")
123.456
```

```
>>> float(5)
5.0
```

`complex()` function

`complex()` can convert an int, a float or a string into a complex.

Examples:

```
>>> complex("1 + 2j")
(1 + 2j)
```

```
>>> complex(5)
(5 + 0j)
```

```
>>> complex(5.1, 6.4)
(5.1 + 6.4j)
```

Data types

`str()` function

`str()` can convert an `int`, a `float` or a `complex` into a `str` (**string**).

Examples:

```
>>> str(123.456)
'123.456'
```

```
>>> str(5)
'5'
```

```
>>> str(1+2j)
'(1+2j)'
```

Comments

Comments

A **comment** is a text in a program that is meant for the human reader; it is **ignored** by the interpreter.

A comment starts with a *hashtag* character `#`.
The interpreter will ignore everyhting from that character to the end of the line.

Example:

```
1 print("This is an example")                #first printed line
2 print("Comments won't be displayed")       #second printed line
```

```
This is an example
Comments won't be displayed
```

Operations

Addition symbol +

- **can** add operands with **same numeric types**
- **can** add operands with **different numeric types**
- **can** *add* **strings** $\rightarrow$ **concatenation**
- **cannot** add a **string with a numeric value** (doing so leads to `TypeError`)

Examples

```
>>> 2 + 3
5
```

```
>>> 1 + 6.4
7.4
```

```
>>> 2.3 + 1.7
4.0
```

```
>>> 'Adding' + ' ' + 'strings'
'Adding strings'
```

Operations

Multiplication symbol *

- **can** multiply operands with **same numeric types**
- **can** multiply operands with **different numeric types**
- **cannot** multiply **strings**
- **can** *multiply* a **string with an integer** $\rightarrow$ **repetition**
  (other numeric types lead to `TypeError`)

Examples

```
>>> 2 * 3
6
```

```
>>> 10 * 6.4
64.0
```

```
>>> 2.5 * 12.2
30.5
```

```
>>> 'Hey' * 3
'HeyHeyHey'
```

Operations

Division symbol /

- **can** divide operands with **same numeric types**
  → always `float` (or `complex`)
- **can** divide operands with **different numeric types**
- **cannot** divide **strings**
- **cannot** divide a **string with an integer** and conversely

Examples

```
>>> 6 / 3
2.0
```

```
>>> 8.32 / 1.3
6.4
```

```
>>> 10 / 2.5
4.0
```

Operations

Integer Division symbol //

Always returns the **integer part** (nearest integer on the left) of the division (can be output on a `float` depending on the operand types)

- **can** be used with `int` or `float` operands (mixed or not)
- **cannot** be used with `str` or `complex` operands

Examples

```
>>> 5 // 2
2
```

```
>>> -5 // 2
-3
```

```
>>> 8.32 // 1.3
6.0
```

```
>>> 10 // 2.5
4.0
```

Operations

Modulo symbol %

Always returns the **remainder** of the division

- **can** be used with `int` or `float` operands (mixed or not)
- **cannot** be used with `str` or `complex` operands

Examples

```
>>> 5 % 2
1
```

```
>>> -5 % 2
1
```

```
>>> 8.32 % 1.3
0.52
```

```
>>> 10 % 2.5
0.0
```

Operations

Exponentiation symbol **

- **can** be used with **any numeric type** operand (mixed or not)
- **cannot** be used with `str` operands

Examples

```
>>> 5 ** 4
625
```

```
>>> 2.5 ** 3.0
15.25
```

```
>>> -5.5 ** 2
-30.25
```

```
>>> 8.32 ** 4.5
13821.493323761144
```

Operations

Operators precedence

Operators have several level of priorities.

List of operators from highest to lowest precedence:

- `**` exponentiation
- `+ -` unary + or - (sign)
- `* / // %` multiplication, division, integer division and modulo
- `+ -` addition and substraction

Parentheses

In order to change the order or operations, you should use **parentheses** to group expressions
Example:

```
>>> 6 + 4 * 5
26
```

```
>>> (6 + 4) * 5
50
```

Operations summary

Operations on numeric values

Symbol	Operation	
`+`	addition	
`-`	substraction	
`*`	multiplication	
`/`	division	always returns a `float` (or `complex`)
`//`	integer division	cannot use `complex`
`%`	modulo (remainder)	cannot use `complex`
`**`	exponentiation	

Operations on strings

Symbol	Operation	
`+`	concatenation	both operands must be `str`
`*`	repetition	one operand must be a `str`, the other one must be an `int`

Variables

What is a variable?

A **variable** is a **name** that refers to a **value**

Once a variable is declared and assigned a value, **you can use that variable where you would use the value**

Example:

```
1 variable_name = 'Hello World!'
2 print(variable_name)
```

```
Hello World!
```

Variables

How do variables actually work ?

```
some_variable_name = "a value"
```

- This is **an assignment statement** - binds a value to a name
- The **equal** sign is the **assignment token** - the "operator" that we use to bind a name to a value
  - name on left (must respect some rules)
  - value on right (can be string, or numeric value, or . . .)

Using variables

When do we use variables?

- when you want your program to *remember* a value
- if you have a value that will *vary*
- examples:
  - a player's score
  - a count of repetitions
  - user input
  - sensor input

Variables

Interactive Shell

Whenever you type something in the shell, it will **always return a value**.

- a value returns a value
- a variable returns a value
- a function can return a value
- if a function does not return a value, the resulting value will be `None` (`NoneType`)

Comments

How to distinguish **something printed** vs **a returned value**?

- returning a string shows the string in quotes, while printing displays the string without quotes
- confusing for other types...

Returned values are not shown in Script mode!!!!

Variables

Variables that are not defined yet

What happens when you use a variable that does not exist?

```
>>> foo
Traceback (most recent call last):
     File "<pyshell#183>", line 1, in <module>
          foo
NameError:  name 'foo' is not defined
>>>
```

Before using any variable, it needs to be **assigned**

Variables

Reassignment

You can reassign or rebind a variable

```
>>> a = 23
>>> a
23
>>> a = "Reassignment"
>>> a
'Reassignment'
```

Variables

Variable Naming Rules

- can be as long as you want
- cannot contain spaces
- consists of **alphanumeric** (numbers and letters) and the **underscore**
- **first character cannot be a number** (only a letter or underscore)
- **case sensitive** (`foo` and `Foo` are 2 different variables)
- cannot be a **keyword** (see Table 1-2 in the textbook p.17)

Exercise

Which of the following are valid variable names in Python?

- `_foo`
- `$foo`
- `foo`
- `3foo`
- `foo bar`
- `#foo`
- `FoO`
- `foobar123`

Variables

Exercise: Using variables

- create a variable called `exclaim`, set it equal to `"!!!"` and print out the variable
- create 2 variables, `length` and `width`
  - set them both equal to `25` and `8` respectively
  - multiply both variables and store the result is a variable called `area`
  - print out the result

Variables

Multiple assignment

It is possible to declare/assign several variables with a single line in Pyhton

```
>>> a, b, c = 3, "Hello", 7
>>> a
3
>>> b
'Hello'
>>> c
7
```

Variables

Exercise: swaping values

We have 2 variables `a` and `b` with values `3` and `21` respectively.
How can we **swap their values** with some code?

Try this on your own...

Variables

Exercise: swaping values

We have 2 variables `a` and `b` with values `3` and `21` respectively. How can we **swap their values** with some code?

First solution: using a third variable for temporary storage

```
1  a = 3
2  b = 21
3  print('a:',a)    #print value of a
4  print('b:',b)    #print value of b
5
6  print()          #blank line
7
8  c = b            #temp variable c
9  b = a
10 a = c
11 print('a:',a)    #print value of a
12 print('b:',b)    #print value of b
```

Output:

```
a:  3
b:  21

a:  21
b:  3
```

Variables

Exercise: swaping values

We have 2 variables `a` and `b` with values `3` and `21` respectively.
How can we **swap their values** with some code?

First solution: using a third variable for temporary storage

Check *Pythontutor* for step-by-step execution:

Link to Pythontutor

- Visualize Execution
- Forward button for executing each line

Variables

Exercise: swaping values

We have 2 variables `a` and `b` with values `3` and `21` respectively. How can we **swap their values** with some code?

Second solution: multiple assignment with Python

```
1  a = 3
2  b = 21
3  print('a:',a)    #print value of a
4  print('b:',b)    #print value of b
5
6  print()          #blank line
7
8  a, b = b, a      #swap values
9  print('a:',a)    #print value of a
10 print('b:',b)    #print value of b
```

Output:

```
a:  3
b:  21

a:  21
b:  3
```

User input

Getting input from the Shell

Programs commonly need to **read input** typed by the user on **keyboard**.

- we can **prompt** the user through the **shell** (console / terminal / command prompt)
- the user enters **input** through the same mechanism

The `input()` function

There is a *builtin* function in Python called `input`

- it *can* take one parameter - the **prompt** that is diplayed
- it returns a **string** representing the user's input (until Enter/Return key is pressed)
- it will **always return a string** - even if the input is a number

User input

Example

```
input('What is your name?')
```

It will display `What is your name?` in the Shell...

...and **wait for a user input**
(keyboard input until Return/Enter key is pressed)

`input` function

How to get/use user's input? $\rightarrow$ **Use a variable!!!**

`input` function returns **a string** that you can **store in a variable**

```
user_name = input('What is your name?')
```

User input

Exercise

Write a program that:

- asks for your name
- and then says "Hi *your name*"

User input

Exercise

Write a program that:

- asks for your name
- and then says "Hi *your name*"

```
1 name = input('What is your name?\n>')
2 print('Hi '+name)
```

```
What is your name?
>Lihua
Hi Lihua
```

User input

Input type

`input` function **always** returns a string.

How do we do if we want to **read numbers** from the user?

→ **Use conversion builtin functions**

```
1  age = int(input('How old are you?\n>'))
2  age = age + 5          #add 5 years
3  print('I thought you were ' + str(age) + ' years old.')
```

```
How old are you?
>20
I thought you were 25 years old.
```

Programming workflow

Steps involved in creating a program

1. Requirements gathering/design
2. Implementation (write the code)
3. Run the program
4. Check output
5. Correct logic errors (go back to step 1)

Which step is the **most important**?

Requirements gathering/design

Design

While you might consider **implementation** is important, it is usually just a **translation of step 1** into a programming language

Requirements gathering/design is:

- understand what your program is supposed to do...
- ...and design how your program will work
  - this is the foundation of your program!
  - can you break down your program into discrete tasks or components?
  - is there an algorithm involved?

Tools for design

- pseudocode
- flowcharts

Design tools

Pseudocode

Sometimes, it is helpful to not have to deal with the *syntax intricacies* and *implementation details* with writing actual code.

- that is where **pseudocode** comes in!
- **pseudocode** is basically *fake* code
- it is more like natural language!
- used to sketch out our actual code...

Example: thermostat program

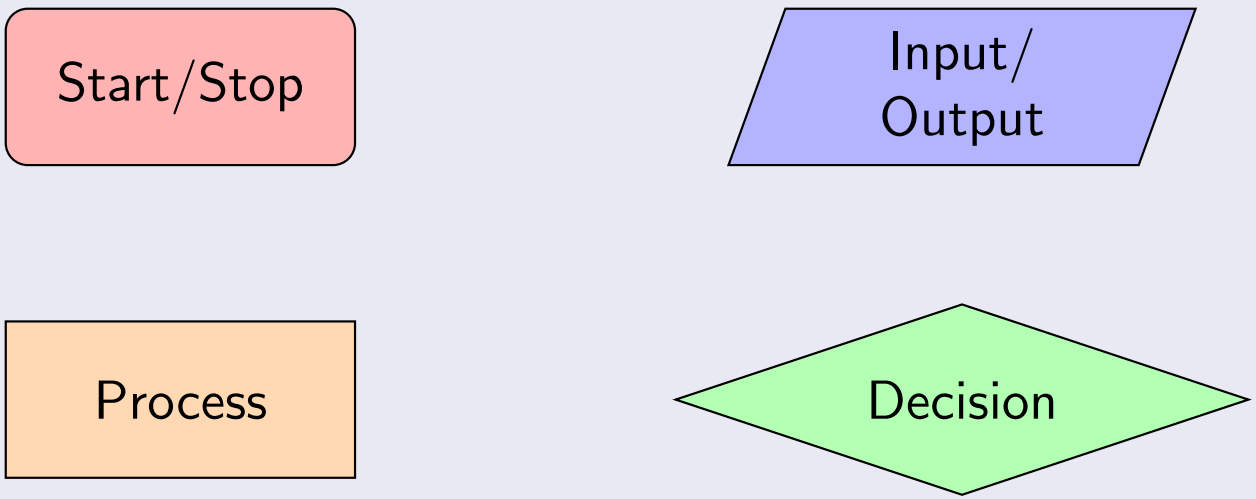
```
measure the temperature of the room

if temperature > theshold
      turn on the air conditioner
else
      turn off the air conditioner
```

Design tools

Flowchart

Flowcharts graphically depict the steps involved in a process or program. Here are the common elements in a flowchart:



Design tools

Example: Fortune telling program

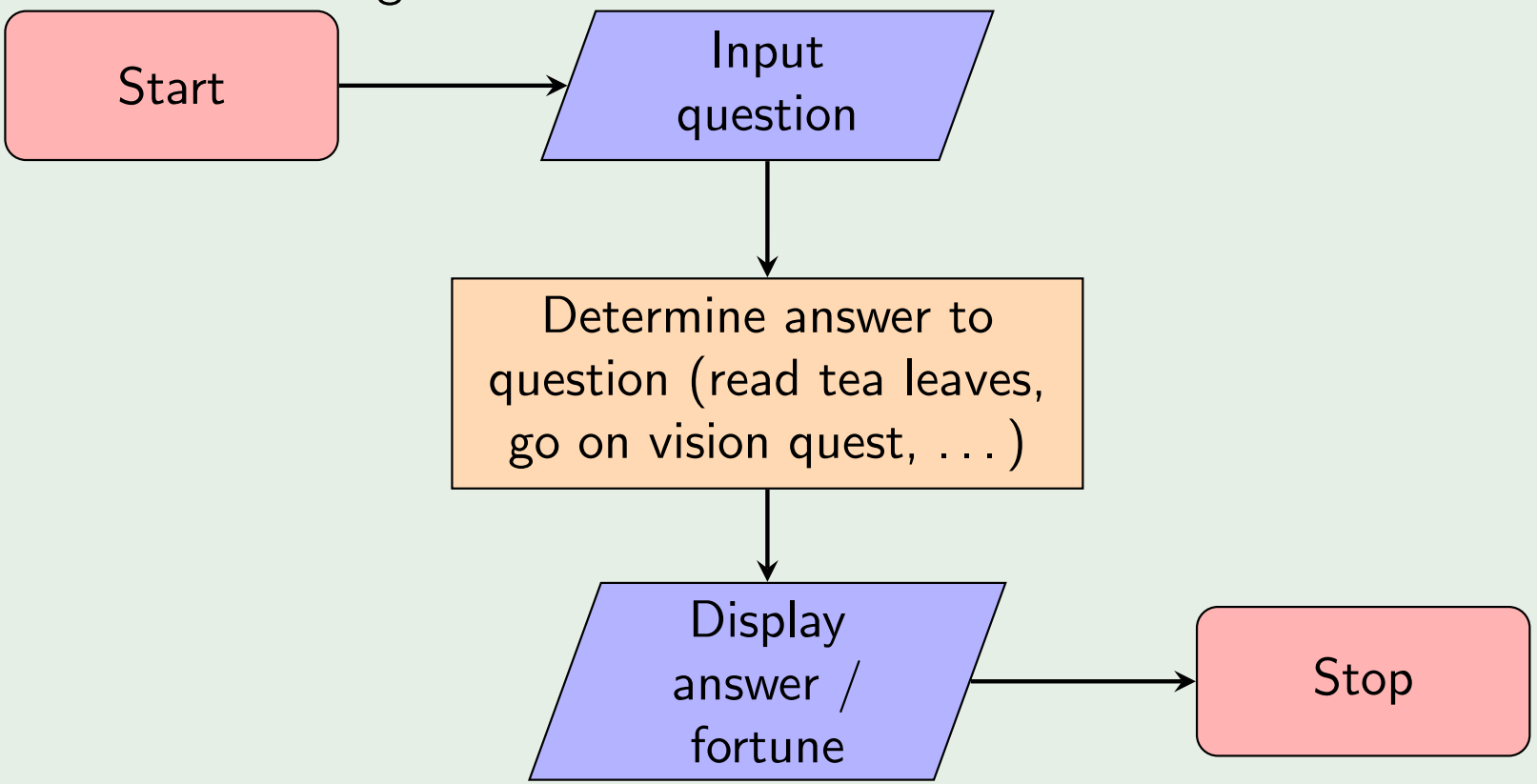
Imagine the following fortune telling program:

```
What is your question?
> Will I have a top grade in ICP?
42
```

Design tools

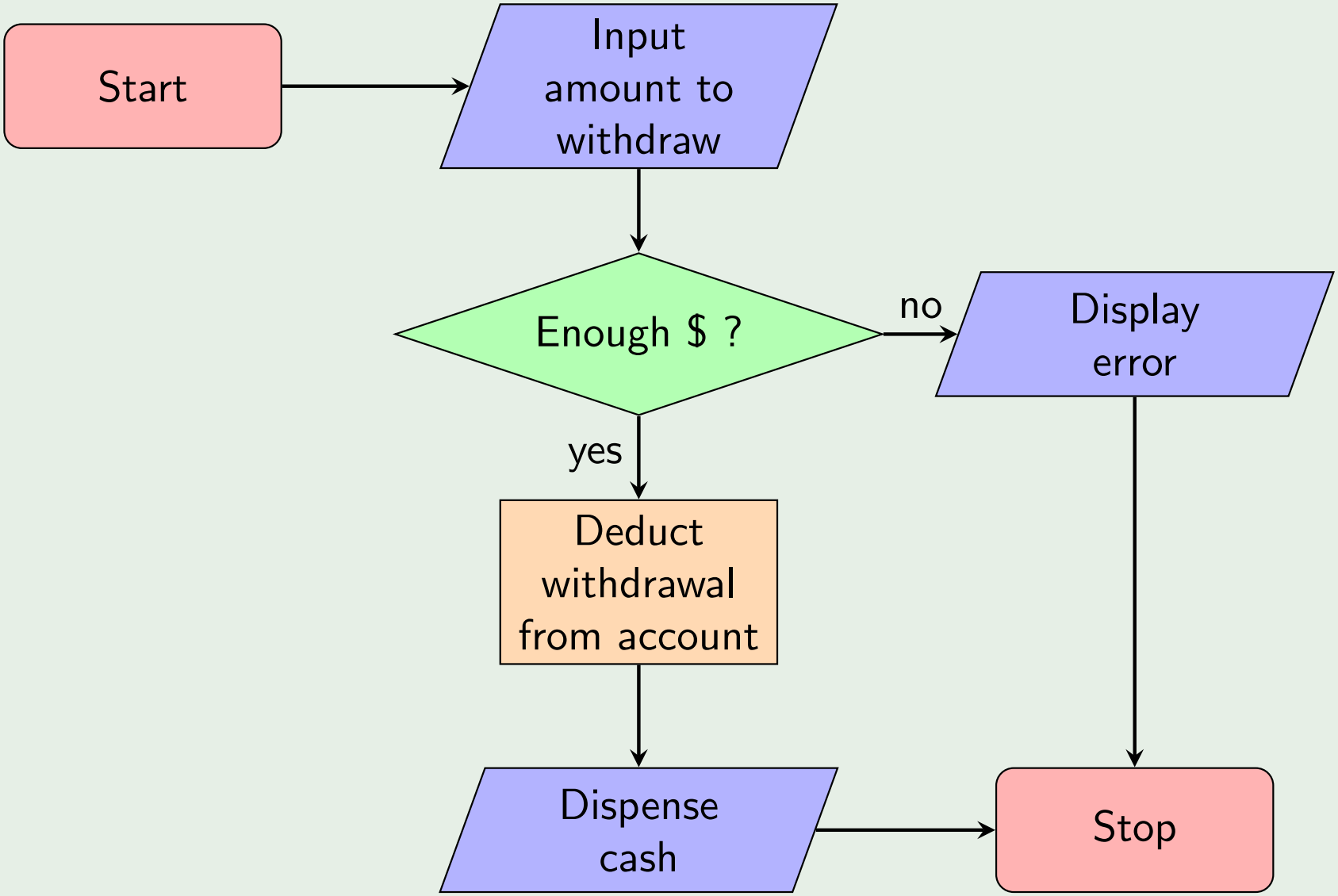
Example: Fortune telling program

The flowchart might look like:



Design tools

Example: ATM program



Design

Input, Processing and Output

The majority of the programs that we write in class will consist of:

- user driven input (usually via keyboard)
- some sort of processing on the input data
- and finally output (usually via Python console)