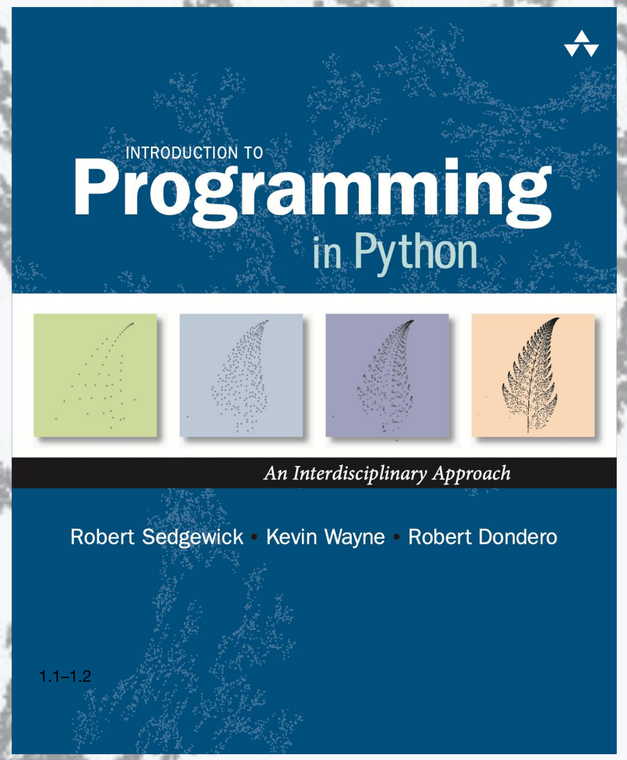
Attendance Register

https://forms.gle/f1UuDkXzQyq47Xvs8

INTRO TO PROGRAMMING IN PYTHON

SEDGEWICK · WAYNE · DONDERO



https://introcs.cs.princeton.edu/python

1. Basic Programming Concepts

INTRO TO PROGRAMMING IN PYTHON

SEDGEWICK • WAYNE • DONDERO

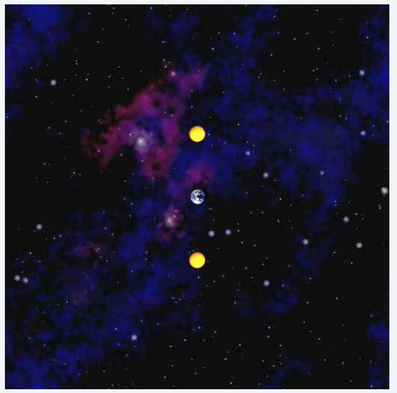
1. Basic Programming Concepts

- Why programming?
- Built-in data types
- Type conversion

Why Programming ?

Why programming? Need to tell the computer what to do.

"Please simulate the motion of N heavenly bodies, subject to Newton's laws of motion and gravity."

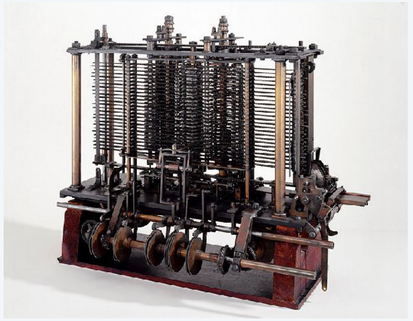


Naive ideal: Natural language instructions.

Prepackaged solutions (apps) are great when what they do is what you want.

Programming enables you to make a computer do anything you want.

Ada Lovelace



Analytic Engine

Programming: telling a computer what to do

Why program ?

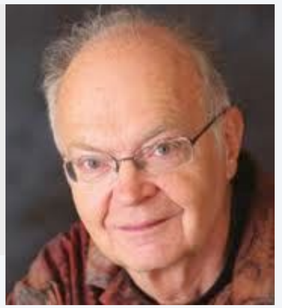
- Is a natural, satisfying and creative experience.
- Enables accomplishments not otherwise possible.
- The path to a new world of intellectual endeavor.

Telling a computer what to do

Challenges

- Need to learn what computers *can* do.
- Need to learn a programming *language*.

" Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. "



– Donald Knuth

Languages

Machine language

- Easy for computer.
- Error-prone for human.

```
10: 8A00    RA ← mem[00]
11: 8B01    RB ← mem[01]
12: 1CAB    RC ← RA + RB
13: 9C02    mem[02] ← RC
14: 0000    halt
```

Adding two numbers

Natural language

- Easy for human.
- Ambiguous and hard for computer to parse

Kids Make Nutritious Snacks.

Red Tape Holds Up New Bridge.

Police Squad Helps Dog Bite Victim.

Actual newspaper headlines

—Rich Pattis

High-level language

- Some difficulty for both.
- An acceptable tradeoff.

```
for (int t = 0; t < 2000; t++)
{
    a[0] = a[11] ^ a[9];
    System.out.print(a[0]);
    for (int i = 11; i > 0; i--)
        a[i] = a[i-1];
}
```

Simulating an LFSR

But *which* high-level language?

Naive ideal: A single programming language for all purposes.

Our Choice: Python

Python features

- Widely used/available.
- Under development since early/mid 1990s.
- Embraces modern abstractions.
- Several widely used libraries (e.g. pandas/scikit-learn)

Guido van Rossum

Facts of life

- No language is perfect.
- You need to start with *some* language.

“ There are only two kinds of programming languages: those people always [gripe] about and those nobody uses.”

– Bjarne Stroustrup

Our approach

- Use a minimal subset of Python.
- Develop general programming skills that are applicable to many languages.

It’s not about the language!

A rich subset of the Python language vocabulary

built-in types
`int`
`float`
`str`
`bool`

operations on numeric types
`+`
`-`
`*`
`/`
`//`
`+=`
`-=`

String *operations*
`+`
`""`
`'`
`len(str)`
`str[i]`

assignment
`=`

flow control
`if`
`else`
`elif`
`for`
`while`

functions / objects
`class`
`def`
`global`
`import`
`from`
`def main()`

Math *methods*
`math.sin()`
`math.cos()`
`math.log()`
`math.exp()`
`math.sqrt()`
`math.pi`

System *methods*
`print()`
`raw_input()`

punctuation
`spaces`
`(`
`)`
`,`
`:`

comparisons
`<`
`<=`
`>`
`>=`
`==`
`!=`

boolean operations
`True`
`False`
`not`
`and`
`or`

arrays
`a[]`
`len(a)`

type conversion methods
`int()`
`float()`
`str()`
`bool()`

our std methods
`stdio.read*()`
`stdio.write*()`
`stdarray.*()`
`stdrandom.*()`
`stddraw.*()`

Your programs will primarily consist of these plus identifiers (names) that you make up.

Hello, World

Requirements

- Python must be installed, VS code and terminal
- ASK DEMI FOR ASSISTANCE DURING TUTORIAL
- https://introcs.cs.princeton.edu/python/home/

Programming in Python

Programming in Python.

- Create the program by typing it into a text editor, and save it as **HelloWorld.py**.

Program 1.1.1 Hello, World (helloworld.py)

```
import stdio

# Write 'Hello, World' to standard output.
stdio.writeln('Hello, World')
```

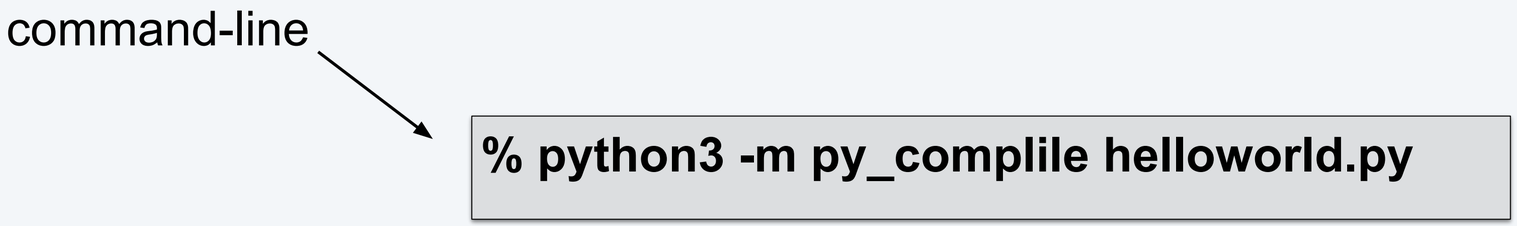
This code is a Python program that accomplishes a simple task. It is traditionally a beginner's first program. The box below shows what happens when you execute the program. The terminal application gives a command prompt (% in this book) and executes the commands that you type (in boldface in this book). The result of executing `python` *with this code is that the program writes* `Hello, World` *in the terminal window (the fourth line).*

```
% python helloworld.py
Hello, World
```

Programming in Python

Programming in Python.

- Create the program by typing it into a text editor, and save it as **HelloWorld.py**.
- **Compile** it by typing in the command-line:
  - python3 -m py_complile helloworld.py.



- This creates a Python bytecode file named: **helloworld.pyc.**

Programming in Python

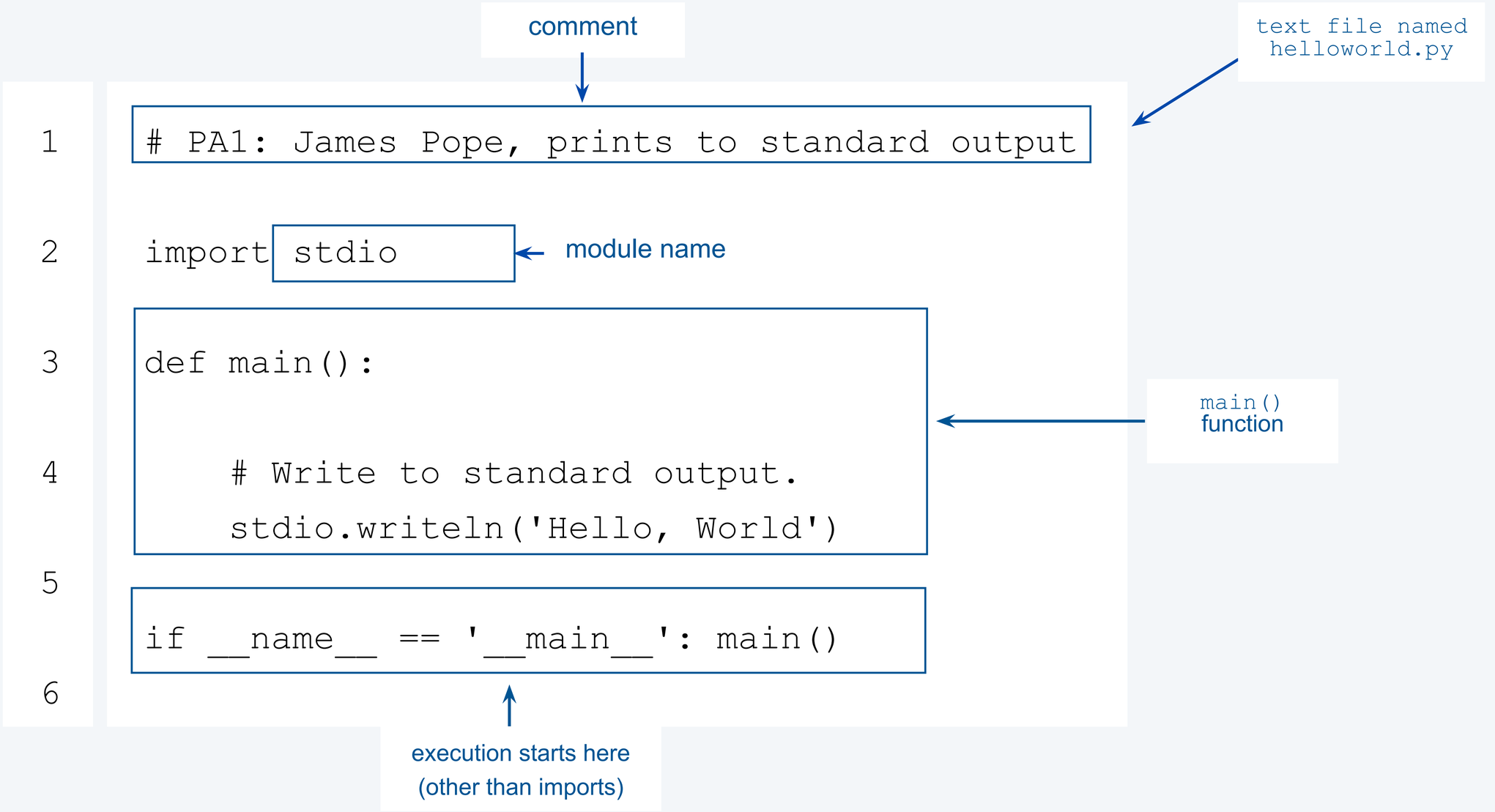
Programming in Python.

- Create the program by typing it into a text editor, and save it as **helloWorld.py**.
- **Compile** it by typing in the command-line:
  - python3 -m py_complile helloworld.py.
- **Execute** it by typing in the command-line:
  - **python3 helloworld.py.**

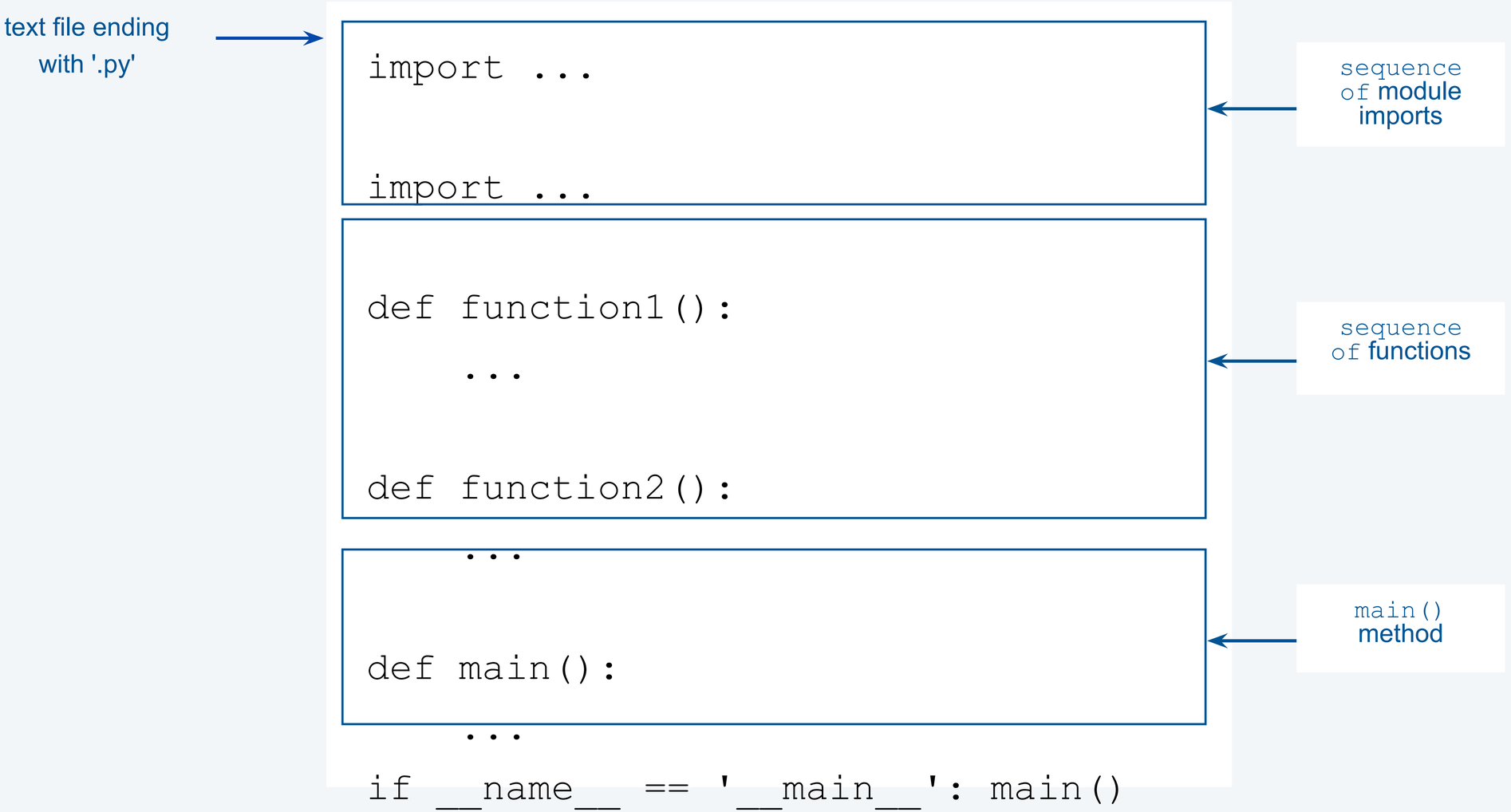
command-line

```
% python3 helloworld.py
Hellow, World
```

Anatomy of your first program



Anatomy of your programs in general



Exercise: programwell.py

Write a program that prints the following line to the terminal: I can program well.

Exercise: Answer

Write a program that prints the following line to the terminal: I can program well.

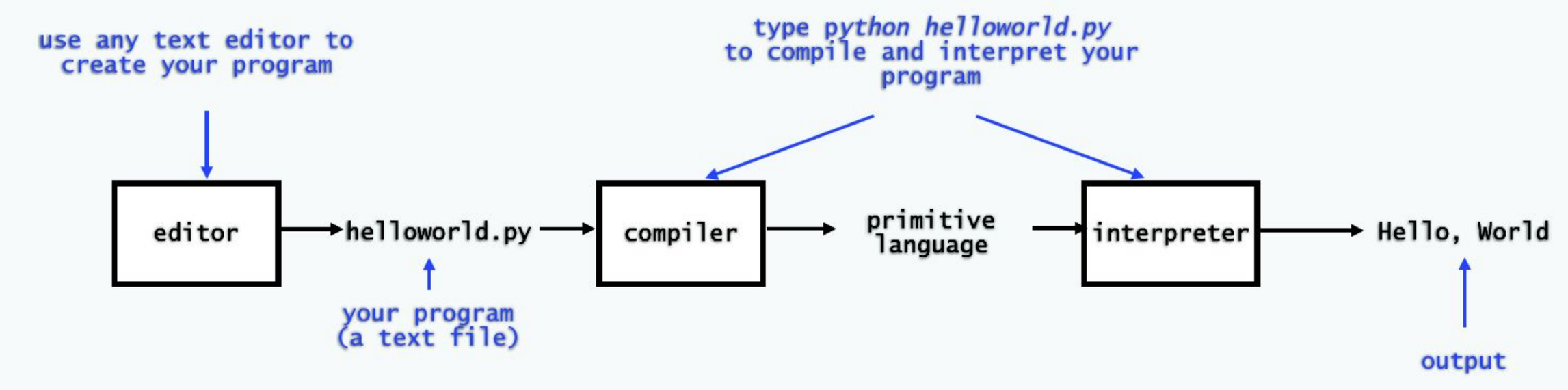
```
import stdio

def main():
    stdio.write("I can program well!")

if __name__ == '__main__': main()
```

programwell.py

Programming in Python: Summary



NB – python3/python is used interchangeably in text

Coding, Compiling and Running

Code refers to program text, while coding refers to the act of creating and editing the code.

A compiler translates a program from the Python language to a language the computer understands (converts a .py file to a .pyc file). It compiles the Python program. The *python compiler* compiles your program.

Executing (or running) is when you transfer control of your computer from system to your program. The *python interpreter* directs your computer to follow your instructions.

Errors

Compile-time errors: caught when the the program is compiled, i.e. a syntax error.

Run-time errors: logical errors in your program which only show up once you execute it.

Errors are also referred to as bugs.

Syntax is important

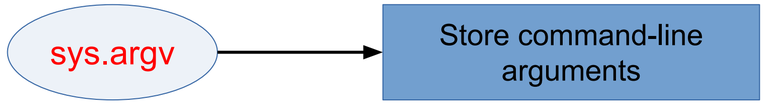

```
A)File "helloworld.py", line 1, in <module>
  import Stdio
  ModuleNotFoundError: No module named 'Stdio'
---
B)File "helloworld.py", line 3
  def main()
           ^
  SyntaxError: invalid syntax
---
C)File "helloworld.py", line 4
  stdio.writeln("Hello, World!")
    ^
  IndentationError: expected an indented block
---
D)File "helloworld.py", line 6, in <module>
  if _name__ == '__main__': main()
  NameError: name '_name__' is not defined
```

UseArgument

Program 1.1.2 Using a command-line argument (useargument.py)

```
import sys
import stdio

stdio.write('Hi, ')
stdio.write(sys.argv[1])
stdio.writeln('. How are you?')
```



This program shows how we can control the actions of our programs: by providing an argument on the command line. Doing so allows us to tailor the behavior of our programs. The program accepts a command-line argument, and writes a message that uses it.

```
% python useargument.py Alice
Hi, Alice. How are you?

% python useargument.py Bob
Hi, Bob. How are you?

% python useargument.py Carol
Hi, Carol. How are you?
```

Exercise: UseArgument 1

Program 1.1.2 Using a command-line argument (useargument.py)

```
import sys
import stdio

stdio.write('Hi, ')
stdio.write(sys.argv[1])
stdio.writeln('. How are you?')
```

sys.argv → Store command-line arguments

What is printed if we modify as follows:
`stdio.writeln(sys.argv[1])`

This program shows how we can control the actions of our programs: by providing an argument on the command line. Doing so allows us to tailor the behavior of our programs. The program accepts a command-line argument, and writes a message that uses it.

```
% python useargument.py Alice
Hi, Alice. How are you?

% python useargument.py Bob
Hi, Bob. How are you?

% python useargument.py Carol
Hi, Carol. How are you?
```

Answer

```
% python useargument.py Alice
Hi, Alice
. How are you?
```

Exercise: UseArgument 2

Program 1.1.2 Using a command-line argument (useargument.py)

```
import sys
import stdio

stdio.write('Hi, ')
stdio.write(sys.argv[1])
stdio.writeln('. How are you?')
```

sys.argv → Store command-line arguments

What is printed if we modify as follows:
`stdio.write(sys.argv[2])`

This program shows how we can control the actions of our programs: by providing an argument on the command line. Doing so allows us to tailor the behavior of our programs. The program accepts a command-line argument, and writes a message that uses it.

```
% python useargument.py Alice
Hi, Alice. How are you?

% python useargument.py Bob
Hi, Bob. How are you?

% python useargument.py Carol
Hi, Carol. How are you?
```

Answer

```
Hi,Traceback (most recent call last):
  File "useargument.py", line 9, in <module>
    if __name__ == '__main__': main()
  File "useargument.py", line 6, in main
    stdio.write(sys.argv[2])
IndexError: list index out of range
```

NB ---> written using a main method

Pop quiz on running "your first program"

Q. Use common sense to cope with the following error messages.

```
% python holamundo.py
python: can't open file 'holamundo.py': [Errno 2] No such file or directory
```

```
% python
Python 2.7.10 (default, Oct  6 2017, 22:29:07)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Pop quiz on running "your first program"

Q. Use common sense to cope with the following error messages.

```
% python holamundo.py
python: can't open file 'holamundo.py': [Errno 2] No such file or directory
```

A. Must have misspelled filename.

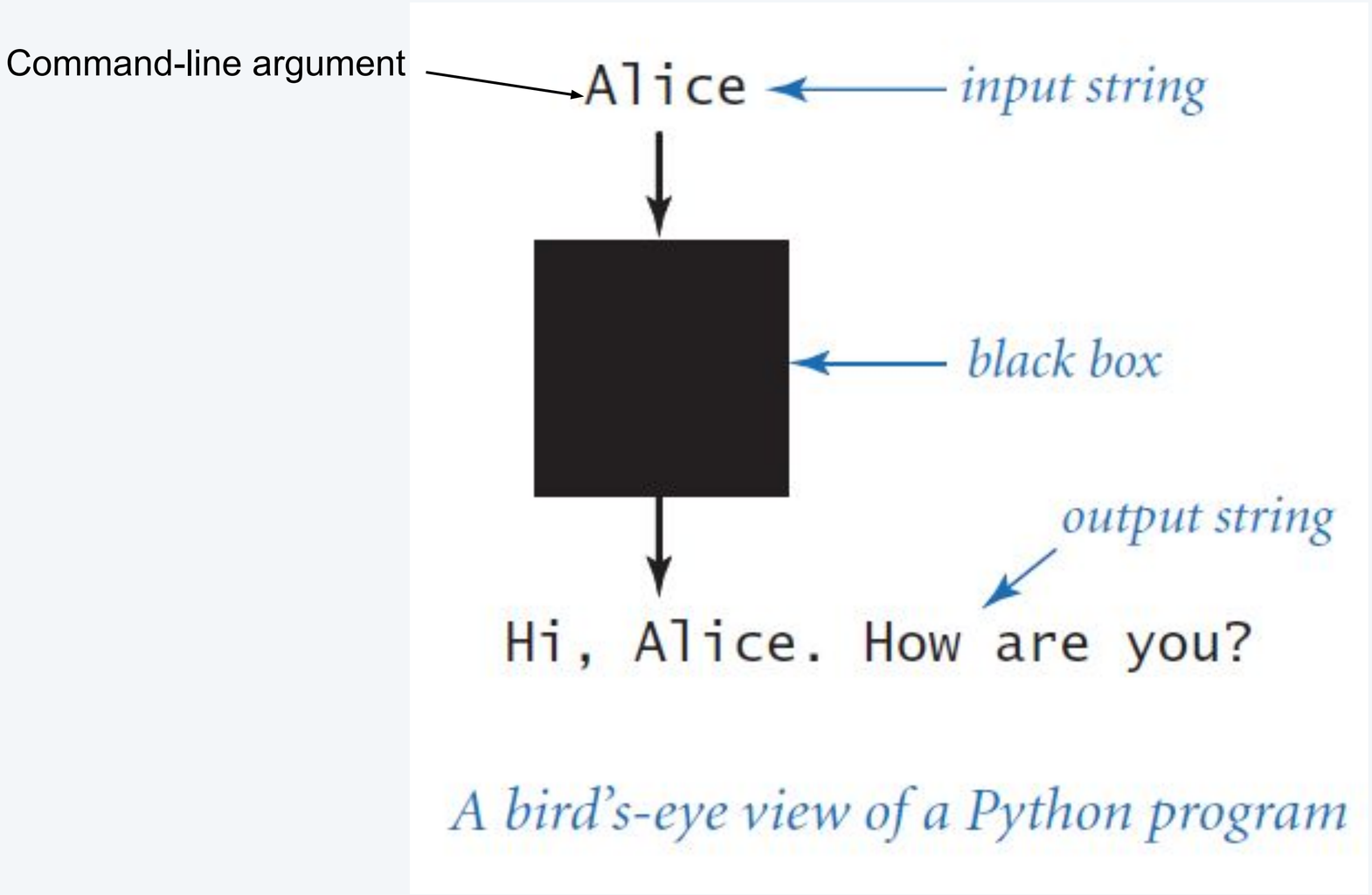
```
python helloworld.py
```

```
% python
Python 2.7.10 (default, Oct  6 2017, 22:29:07)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

A. Oops, forgot to pass python file.

```
python helloworld.py
```

Input and Output



A bird's-eye view of a Python program

Algorithmic Thinking

Algorithm

Algorithm: a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

Example

Develop an algorithm to pick the largest number from the following list
1, 7, 9, 3, 5, 6

1. Choose first number and assign to it the label of current number

2. Compare the remaining numbers with the current number in the order given in the list.

3. If a larger number is found make it the current number and then continue comparing as before (continue where you are in the list).

4. Continue in this way until you have run through all the numbers in the list.

5. The number labelled as current number will be the largest number in the list.

Exercise: Puzzle

Develop an algorithm which can be used to complete a jigsaw puzzle. You may assume the puzzle has no missing pieces. You may assume that you can clearly identify the top, left, bottom and right hand side of each piece.

Exercise: Puzzle

Develop an algorithm which can be used to complete a jigsaw puzzle. You may assume the puzzle has no missing pieces. You may assume that you can clearly identify the top, left, bottom and right hand side of each piece.

1. Put first piece down or choose a random piece which is already connected to the puzzle and still has unconnected sides. We will call this piece the focus piece. If there are no such pieces the puzzle is complete.

2. Pick the first unconnected side along a clockwise direction (start at its left side) of the focus piece and label it side 1. If there are no more unconnected sides go to step 1.

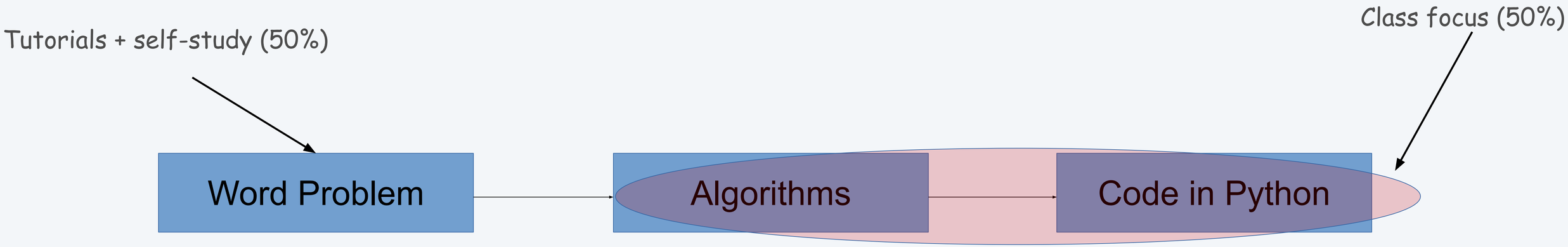
3. Pick up a random unconnected puzzle piece (discarding pieces you have tried before in conjunction with the current focus piece) and label it temporary piece.

4. Pick the first open side along a clockwise direction (start at its left side) of the temporary piece discarding sides that you have already tried (with the current focus-temporary pair) and label it side 2. If you have tried all sides put the piece down and go to step 3.

5. Compare side 1 and side 2. If they fit go to step 2 else go to step 4.

Problem Solving

Problem Solving. Word problems, which you need to convert to an algorithm which you then have to code up.



INTRO TO PROGRAMMING IN PYTHON

SEDGEWICK • WAYNE • DONDERO

1. Basic Programming Concepts

- Why programming?
- Built-in data types
- Type conversion

Built-in data types

A data type is a set of values and a set of operations on those values.

type	*set of values*	*sample literals*	*operations*
`str`	sequences of characters	`'Hello World'` `"CS isn't fun"`	+
`int`	integers	`17` `12345`	+ - * // % **
`float`	floating-point numbers	`3.1415` `6.022e23`	+ - * // % **
`bool`	truth values	`True` `False`	and, or, not

Python's built-in data types

Pop quiz on data types

Q. What is a data type?

Pop quiz on data types

Q. What is a data type?

A. A set of values and a set of operations on those values.

Basic Definitions

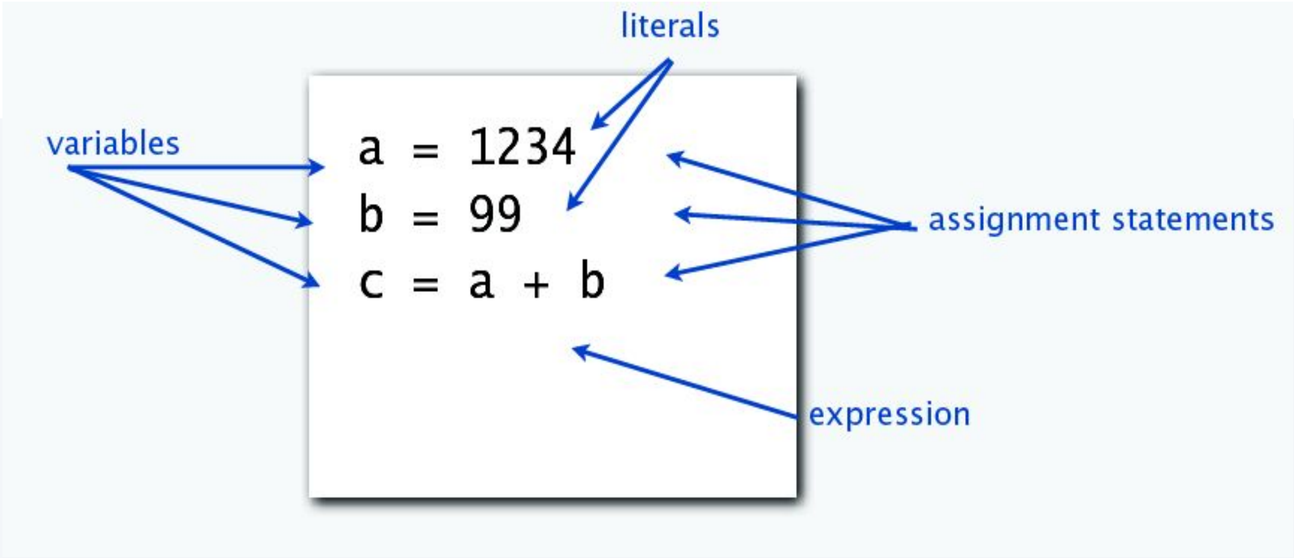
Variable is a name that refers to a data-type value.

Identifiers is a Python-code representation of a name.

Literal is a programming-language representation of a data-type value.

Assignment statement using the = operator associates a data-type value with a variable.

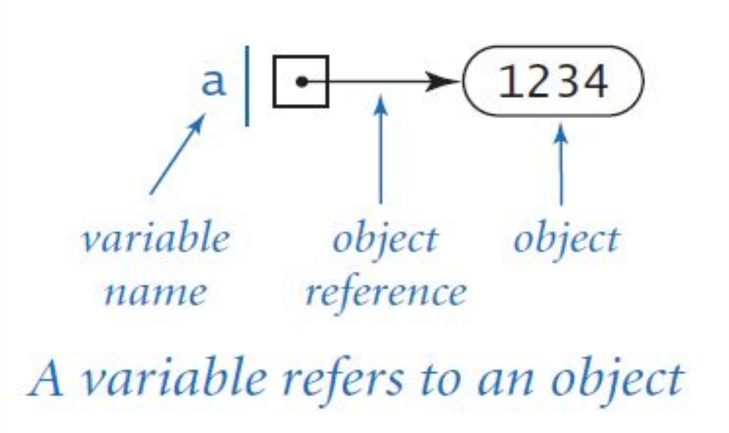
Expression is a combination of literals, variables, and operators that Python evaluates to produce a value.



The meaning of = in a program is NOT the same as in a mathematical equation.

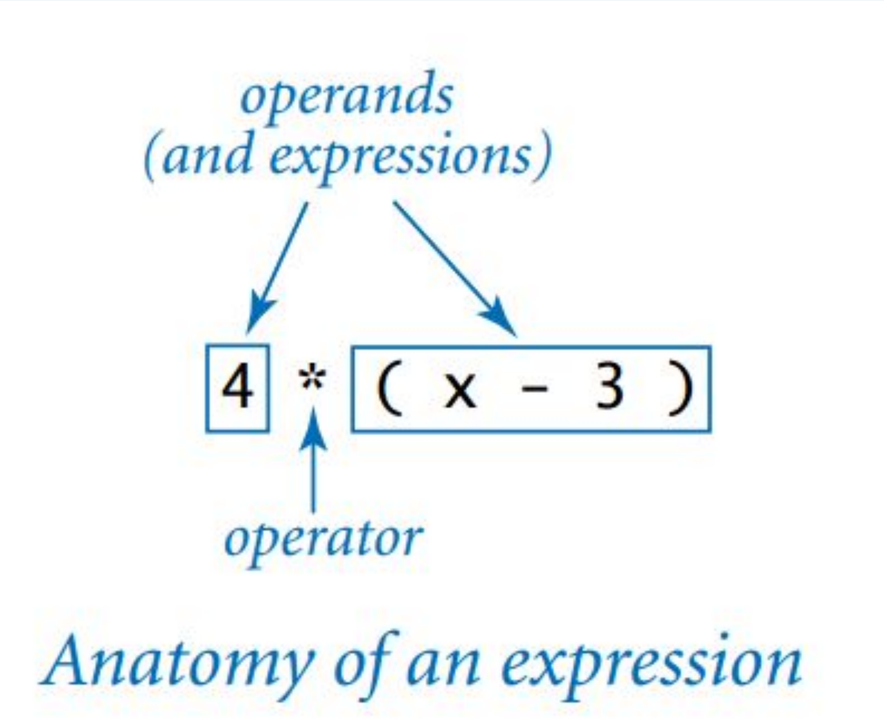
Objects

- Object. An object is an in-computer-memory representation of a value from a particular data type.
  - Identity. Uniquely identifies an object (memory address)
  - Type. The set of values it might represent and the set of operations that can be performed on it.
  - Value. The data-type value that it represents
- Object reference. An object reference is nothing more than a concrete representation of the object's identity (memory address).



A variable refers to an object

Expressions

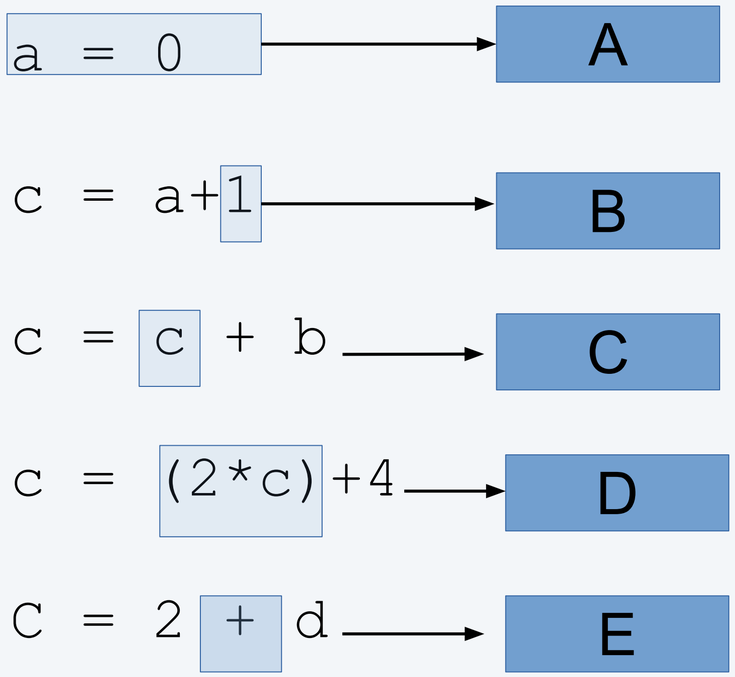


Anatomy of an expression

Operators specify data type operations on one or more operand.
Operand can be any expression.
Binary operators take two expressions.
Precedence multiplication and division before addition and subtraction.
Left-associativity a-b-c same as (a-b)-c. All operators except exponentiation.
Right-associativity a**b**c same as a**(b**c).
Parenthesis overrides precedence rules.

Exercise: Definitions

Fill in the missing labels:



Answer

Fill in the missing labels:

```
A → Assignment
B → Literal/Operand
C → Variable/Operand
D → Expression
E → Operator
```

Object Trace

Program behavior is typically understood using traces.

Informal trace table showing values after each line of the program.

Object trace shows variables, object references, and objects for each line of the program. An object trace example is below.

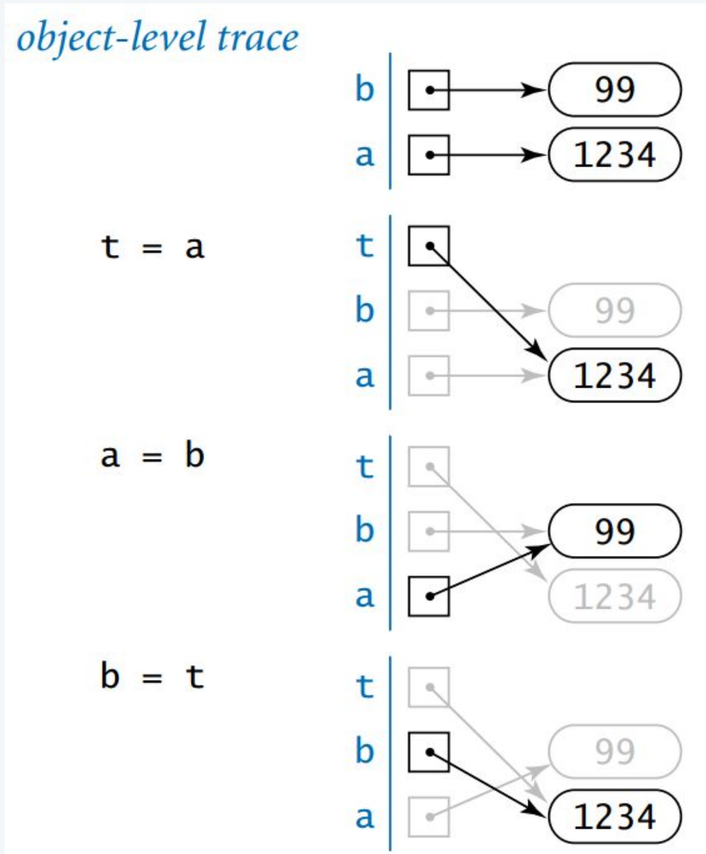
informal trace

	a	b	t
	1234	99	
t = a	1234	99	1234
a = b	99	99	1234
b = t	99	1234	1234

```
def main():
    a = 1234
    b = 99
    t = a
    a = b
    b = t
if __name__ == '__main__': main()
```

This code *exchanges* the values of a and b.

object-level trace



Basic Conversions

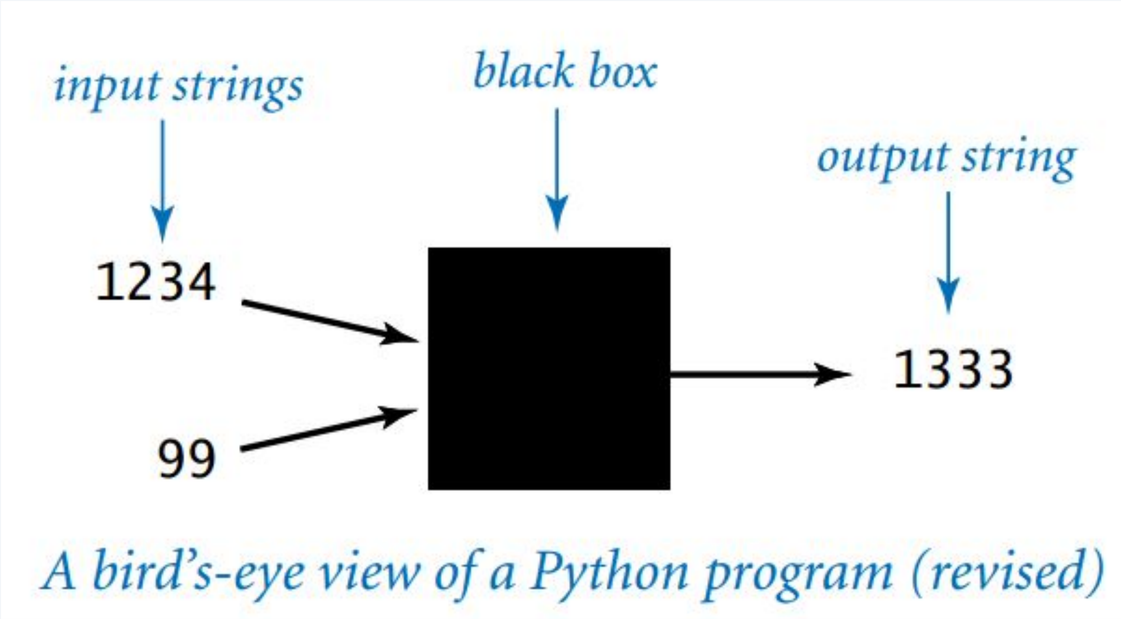
str-to-int

```
int(“123”)
int(sys.argv[1])
```

int-to-str

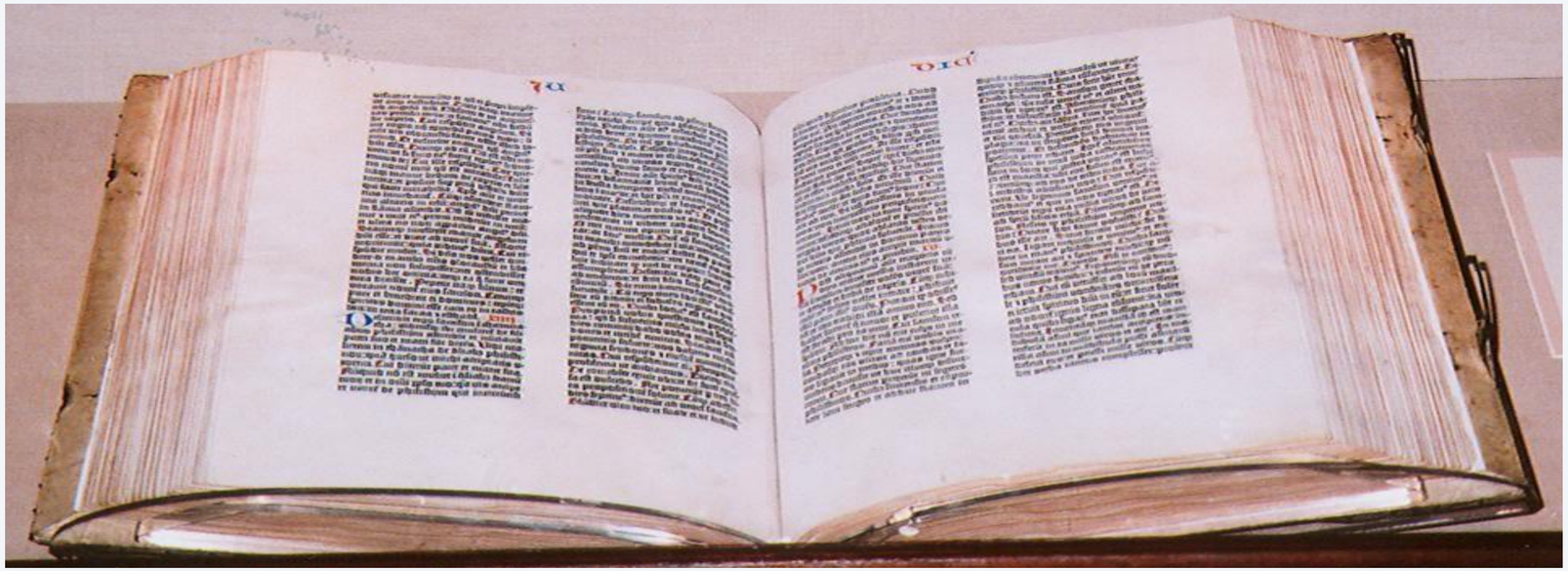
```
a = “123”
b = 99
c = int(a) + b
stdio.writeln(a+” + “+str(b)+” = ”+str(c))
```

123 + 99 = 222



A bird’s-eye view of a Python program (revised)

Text



Data type for computing with strings: str (will use str and string)

`str` data type

values	sequences of characters
typical literals	`'Hello, '  '1 '  ' * '`
operation	concatenate
operator	`+`
escape sequence	`'\t', '\n', '\\', and '\''`

Examples of `String` operations (concatenation)

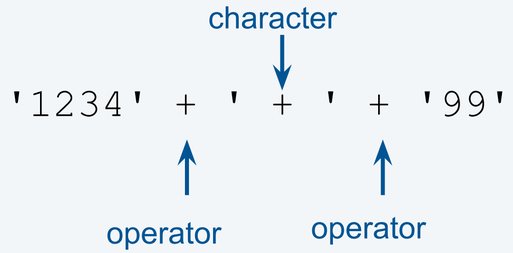
expression	*value*
'Hi, ' + 'Bob'	'Hi, Bob'
'1' + ' 2 ' + '1'	'1 2 1'
'1234' + ' + ' + '99'	'1234 + 99'
'1234' + '99'	'123499'

Typical use: Input and output.

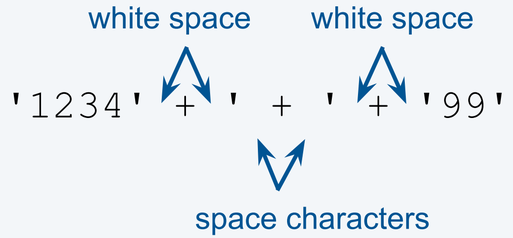
Important note:

Character interpretation depends on context!

Ex 1: plus signs



Ex 2: spaces



Example of computing with strings: subdivisions of a ruler

```
import stdio

ruler1 = '1'
ruler2 = ruler1 + ' 2 ' + ruler1
ruler3 = ruler2 + ' 3 ' + ruler2
ruler4 = ruler3 + ' 4 ' + ruler3
stdio.writeln(ruler1)
stdio.writeln(ruler2)
stdio.writeln(ruler3)
stdio.writeln(ruler4)
```

all + ops are concatenation

1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

```
% python3 Ruler.py
1
1 2 1
1 2 1 3 1 2 1
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

	ruler1	ruler2	ruler3	ruler4
	undeclared	*undeclared*	*undeclared*	*undeclared*
`ruler1 = '1'`	1	undeclared	*undeclared*	*undeclared*
`ruler2 = ruler1 + ' 2 ' + ruler1`	1	1 2 1	*undeclared*	*undeclared*
`ruler3 = ruler2 + ' 3 ' + ruler2`	1	1 2 1	1 2 1 3 1 2 1	*undeclared*
`ruler4 = ruler3 + ' 4 ' + ruler3`				1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

Exercise: Ruler

What does the following program print to the terminal?

```
import sys
import stdio

def main():
    ruler1 = "x"
    ruler2 = ruler1 + " * " + ruler1
    ruler3 = ruler2 + " # " + ruler2
    ruler4 = ruler3 + " * " + ruler3
    stdio.writeln(ruler4)

if __name__ == "__main__": main()
```

Answer

What does the following program print to the terminal?

```
import sys
import stdio

def main():
    ruler1 = "x"
    ruler2 = ruler1 + " * " + ruler1
    ruler3 = ruler2 + " # " + ruler2
    ruler4 = ruler3 + " * " + ruler3
    stdio.writeln(ruler4)

if __name__ == "__main__": main()
```

```
"x"
"x * x"
"x * x # x * x"
```

string concatenation

```
% python3 rulermodified.py
x * x # x * x * x * x # x * x
```

Integers

…, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, …

Integers

int data type represents integers or natural numbers.

values	Python3 only limited by memory Python2 has int (2^{63}-1) and long (arbitrary)					
typical literals	1234	99	0	1000000		
operations	add	subtract	multiply	quotient	remainder	power
operator	+	-	*	//	%	**

floored divide

Also unary *sign* operator
+ and –

Examples of int operations

expression	*value*	*comment*
1234 + 5	1239	
1234 - 5	1229	
1234 * 5	6170	
1234 // 5	246	*drop fractional part*
1234 % 5	4	*remainder*
1234 ** 5	*28613817210 51424*	

Precedence

expression	*value*	*comment*
3 * 5 - 2	13	** has precedence*
3 + 5 // 2	5	*// has precedence*
3 - 5 - 2	-4	*left associative*
4 ** 3 ** 2	262144	*right associative*
(3 - 5) - 2	-4	*better style*
4 ** (3 ** 2)	262144	*better style*

use parentheses to avoid these rules

Typical usage: Math calculations; specifying programs (stay tuned).

Integer Operations

```
import sys
import stdio

def main():
   a = int(sys.argv[1])
   b = int(sys.argv[2])
   total  = a + b
   diff   = a - b
   prod   = a * b
   quot   = a // b
   rem    = a % b
   exp    = a ** b

   stdio.writeln(str(a) + ' + ' + str(b) + ' = ' + str(total))
   stdio.writeln(str(a) + ' - ' + str(b) + ' = ' + str(diff))
   stdio.writeln(str(a) + ' * ' + str(b) + ' = ' + str(prod))
   stdio.writeln(str(a) + ' // ' + str(b) + ' = ' + str(quot))
   stdio.writeln(str(a) + ' % ' + str(b) + ' = ' + str(rem))
   stdio.writeln(str(a) + ' ** ' + str(b) + ' = ' + str(exp))

if __name__ == '__main__': main()
```

command-line arguments

convert int to str

```
% python3 intops.py 5 2
5 + 2 = 7
5 - 2 = 3
5 * 2 = 10
5 // 2 = 2
5 % 2 = 1
5 ** 2 = 25

% python3 intops.py 1234 5
1234 + 5 = 1239
1234 - 5 = 1229
1234 * 5 = 6170
1234 // 5 = 246
1234 % 5 = 4
1234 ** 5 = 2861381721051424
```

1234 = 246*5 + 4

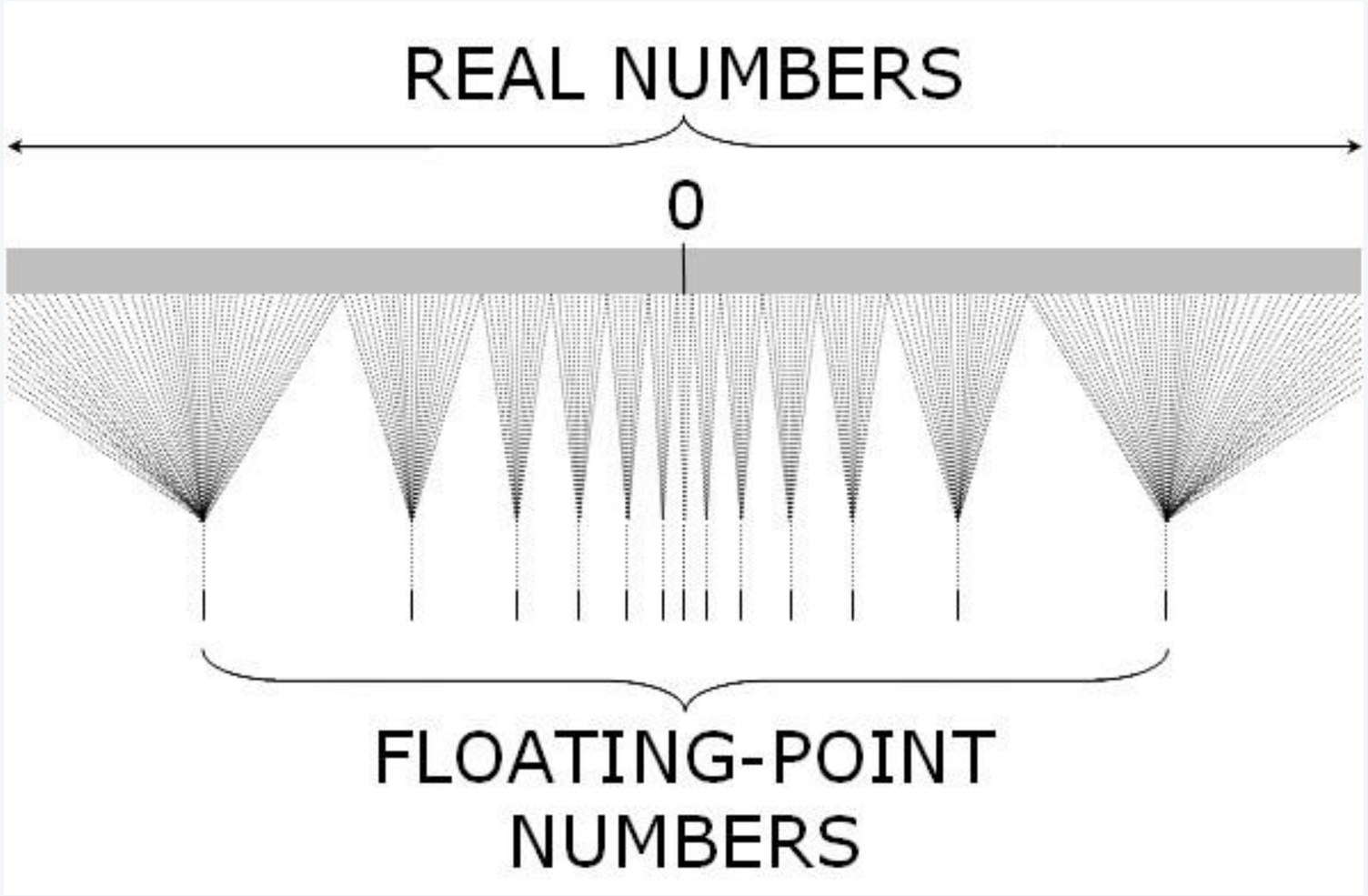
Excercise: intops

```
% python3 intops.py 111 3
```

Answer

```
% python3 intops.py 111 3
111 + 3 = 114
111 - 3 = 108
111 * 3 = 333
111 // 3 = 37
111 % 3 = 0
111 ** 3 = 1367631
```

Floating-Point Numbers



Floating-Point Numbers

`float` data type

https://docs.python.org/3.2/tutorial/floatingpoint.html

values	real numbers (sys.float_info, IEEE 754)				
typical literals	3.14159	2.0	1.4142135623730951	6.022e23	
operations	add	subtract	multiply	divide	exponentiation
operator	+	-	*	/	**

6.022×10^{23}

Typical float values are *approximations*

Examples:

no `float` value for π.

no `float` value for $\sqrt{2}$

no `float` value for 1/3.

Examples of `float` operations

expression	*value*
3.141 + .03	3.171
3.141 - .03	3.111
6.02e23/2	3.01e23
5.0 / 3.0	1.6666666666666667
10.0 % 3.141	0.577
math.sqrt(2.0)	1.4142135623730951

Scientific notation

Typical use: Scientific calculations.

10.0-3*3.141

Special expressions/values

expression	*value*
1.0 / 0.0	ZeroDivisionError
math.sqrt(-1.0)	ValueError
math.inf	+ infinity
math.nan	not a number

undefined

Exercise: Mod

Compute 10.0%0.62e1

Exercise: Mod

Compute 10.0%0.62e1

3.8 = 10.0-1*6.2

Excerpts from Python's Math Library

built-in functions	
abs(a)	*absolute value of a*
max(a,b)	*maximum value of a and b*
min(a,b)	*maximum value of a and b*
standard functions from math module	
sin(theta)	*sine function*
cos(theta)	*cosine function*
tan(theta)	*tangent function*
exp(a)	*exponential (e^a)*
log(a)	*natural log ($\log_e$ a, or ln a)*
pow(a, b)	*raise a to the b'th power (a^b)*
sqrt(a)	*square root of a*
e	*value of e (constant)*
pi	*value of π (constant)*
standard functions from random	
random.random	*float in [0,1)*
random.randrange(x,y)	*int in [x,y)*

inverse functions also available: asin(), acos(), and atan()

theta in radians. Use degrees() and radians()) to convert.

You can discard your calculator now (please).

Quadratic Equation

From algebra: the roots of $x^2 + bx + c$ are $\frac{-b \pm \sqrt{b^2 - 4c}}{2}$

```
import sys
import stdio
import math

def main():
   b = float(sys.argv[1])
   c = float(sys.argv[2])
   discriminant = b*b - 4.0*c
   d = math.sqrt(discriminant)
   stdio.writeln((-b + d) / 2.0)
   stdio.writeln((-b - d) / 2.0)

if __name__ == '__main__': main()
```

```
% python3 quadratic.py -3.0 2.0
2.0
1.0

% python3 quadratic.py -1.0 -1.0
1.61803398875
-0.61803398875

% python3 quadratic.py 1.0 1.0
ValueError: math domain error

% python3 quadratic.py 1.0 hello
ValueError: could not convert string to float

% python3 quadratic.py 1.0
IndexError: list index out of range
```

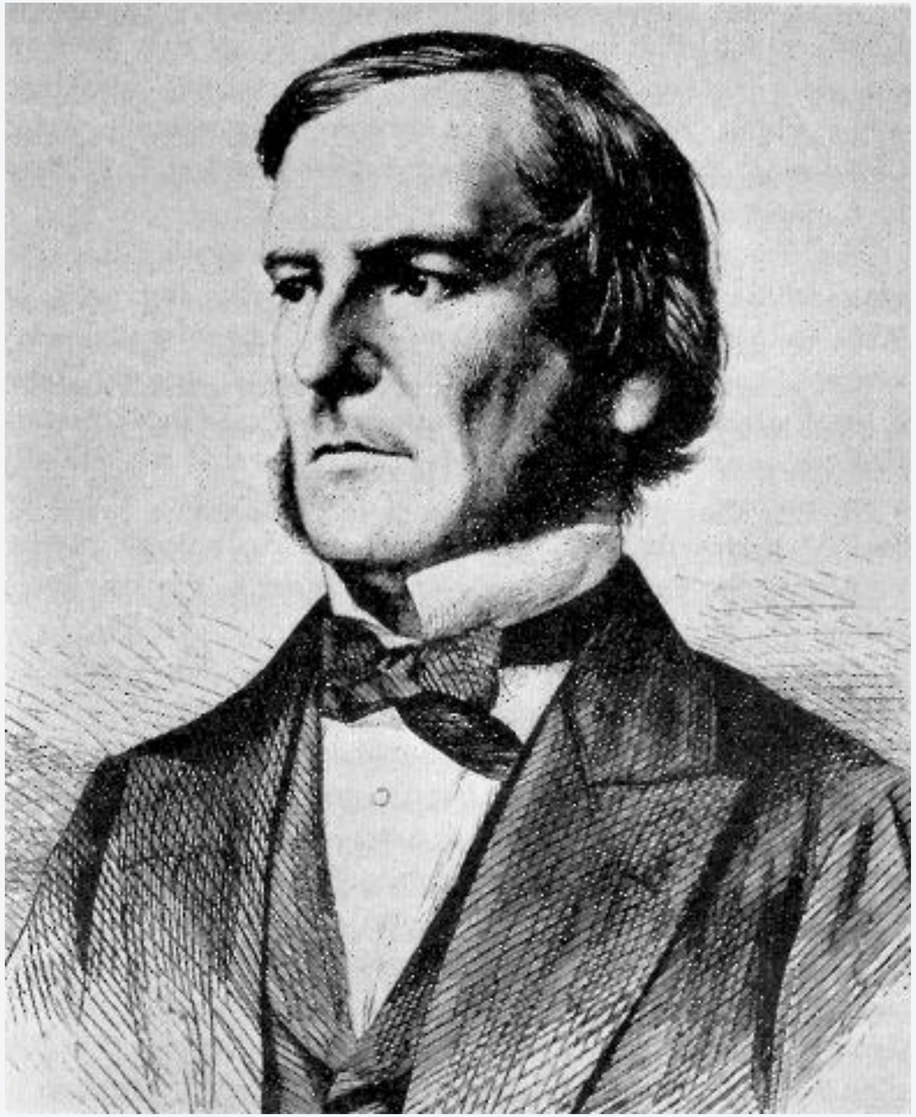
$x^2 - 3x + 2$

$x^2 - x - 1$

$x^2 + x + 1$

Need two arguments.
(Fact of life: Not all error messages are crystal clear.)

Booleans



Booleans

- **boolean data type** represents truth values (either true or false) from logic.

bool data type

logical values	true	false	
literals	True	False	
operations	and	or	not
a.k.a.	&&	||	!

Truth-table definitions

a	not a	a	b	a and b	a or b
true	false	false	false	false	false
false	true	false	true	false	true
		true	false	false	true
		true	true	true	true

Python shortcut a ^ b

Q. a XOR b?

A. (not a and b) or (a and not b)

Proof

a	b	(not a and b)	(a and not b)	(not a and b) or (a and not b)
false	false	false	false	false
false	true	true	false	true
true	false	false	true	true
true	true	false	false	false

Typical usage: Control logic and flow of a program (used in if and while statements, Section 1.3).

Exercise: Booleans

What does the following program print to the terminal when executed?

```
import sys
import stdio

def main():
    x = True
    y = False
    x = (x and y) or not(x or y)
    y = (x and y) or (x or y)
    stdio.write(str(x))
    stdio.write(" "+str(y))
    stdio.writeln(" "+str(x ^ y))

if __name__ == "__main__": main()
```

Answer

What does the following program print to the terminal when executed?

- stdio.write(str (x)) -> False
- stdio.write (" "+ str (y)) -> False
- stdio.write (" "+ str (x ^ y)) -> False

```
x = (x and y) or not(x or y)

x = (true and false) or not(true or false)

x = (false) or not(true)

x = false or false

x = false

y = (x and y) or (x or y)

y = (false and false) or (false or false)

y = (false) or (false)

y = false

false = false xor false
```

Comparisons

Comparisons. Take two operands of one type (e.g., **int)** and produce a result of type **bool**. Lower precedence than arithmetic operators, higher than boolean operators.

op	*meaning*	true	false
==	*equal*	`2 == 2`	`2 == 3`
!=	*not equal*	`3 != 2`	`2 != 2`
<	*less than*	`2 < 13`	`2 < 2`
<=	*less than or equal*	`2 <= 2`	`3 <= 2`
>	*greater than*	`13 > 2`	`2 > 13`
>=	*greater than or equal*	`3 >= 2`	`2 >= 3`

non-negative discriminant?	`(b*b - 4.0*a*c) >= 0.0`
beginning of a century?	`(year % 100) == 0`
legal month?	`(month >= 1) && (month <= 12)`

Leap Year

Q. Is a given year a leap year?

A. Yes if either (i) divisible by 400 or (ii) divisible by 4 but not 100.

Program 1.2.5 Leap year (leapyear.py)

```
import sys
import stdio

year = int(sys.argv[1])

isLeapYear = (year % 4 == 0)
isLeapYear = isLeapYear and ((year % 100) != 0)
isLeapYear = isLeapYear or  ((year % 400) == 0)

stdio.writeln(isLeapYear)
```

This program tests whether an integer corresponds to a leap year in the Gregorian calendar. A year is a leap year if it is divisible by 4 (2004), unless it is divisible by 100 in which case it is not (1900), unless it is divisible by 400 in which case it is (2000).

```
% python leapyear.py 2016
True
% python leapyear.py 1900
False
% python leapyear.py 2000
True
```

Example: Leap Year

((1900%4 == 0) && (1900 % 100 != 0)) || (1900 % 400 == 0)

=> ((0 == 0) && (0 != 0)) || (300 == 0)

=> (true && false) || false

=> false

```
% python3 leapyear.py 1900
false
```

Exercise: Leap Year

```
((2000%4  == 0) && (2000 % 100 != 0)) || (2000 % 400 == 0)

COMPLETE....
```

```
% python3 leapyear.py 2000
true
```

Answer

((2000%4 == 0) && (2000 % 100 != 0)) || (2000 % 400 == 0)

=> ((0 == 0) && (0 != 0)) || (0 == 0)

=> (true && false) || true

=> true

```
% python3 leapyear.py 2000
true
```

Library Methods and API

Library

Library. Existing Python methods (module) written by someone else (grouped by topic).
API. Applications Programming Interface. Table summarizing module methods.

built-in functions

`abs(x)`	*absolute value of* x
`max(a, b)`	*maximum value of* a *and* b
`min(a, b)`	*minimum value of* a *and* b

booksite functions for standard output from our `stdio` *module*

`stdio.write(x)`	*write* x *to standard output*
`stdio.writeln(x)`	*write* x *to standard output, followed by a newline*

Note 1: Any type of data can be used (and will be automatically converted to `str`*).*
Note 2: If no argument is specified, x *defaults to the empty string.*

standard functions from Python's `random` *module*

`random.random()`	*a random* `float` *in the interval [0, 1)*
`random.randrange(x, y)`	*a random* `int` *in [*x, y*) where* x *and* y *are* `int`s

standard functions from Python's `math` *module*

`math.sin(x)`	*sine of* x *(expressed in radians)*
`math.cos(x)`	*cosine of* x *(expressed in radians)*
`math.tan(x)`	*tangent of* x *(expressed in radians)*
`math.atan2(y, x)`	*polar angle of the point* (x, y)
`math.hypot(x, y)`	*Euclidean distance between the origin and* (x, y)
`math.radians(x)`	*conversion of* x *(expressed in degrees) to radians*
`math.degrees(x)`	*conversion of* x *(expressed in radians) to degrees*
`math.exp(x)`	*exponential function of* x (e^x)
`math.log(x, b)`	*base-*b *logarithm of* x ($\log_b x$) *(the base* b *defaults to* e*—the natural logarithm)*
`math.sqrt(x)`	*square root of* x
`math.erf(x)`	*error function of* x
`math.gamma(x)`	*gamma function of* x

Note: The `math` *module also includes the inverse functions* `asin()`, `acos()`, *and* `atan()` *and the constant variables* `e` *(*`2.718281828459045`*) and* `pi` *(*`3.141592653589793`*).*

More...

Arguments. Inputs enclosed in parentheses and separated by commas following function name. Python *calls* (or evaluates) the function with the given arguments and the function then *returns* a value.

pure function – consistent/no side effects

built into Python

function call	*return value*	*comment*
`abs(-2.0)`	`2.0`	*built-in function*
`max(3, 1)`	`3`	*built-in function with two arguments*
`stdio.write('Hello')`		*booksite function (with side effect)*
`math.log(1000, math.e)`	`6.907755278982137`	*function in* `math` *module*
`math.log(1000)`	`6.907755278982137`	*second argument defaults to* `math.e`
`math.sqrt(-1.0)`	*run-time error*	*square root of a negative number*
`random.random()`	`0.3151503393010261`	*function in* `random` *module*

Typical function calls

standard function – part of standard Python modules

output other than return value

INTRO TO PROGRAMMING IN PYTHON

SEDGEWICK • WAYNE • DONDERO

1. Basic Programming Concepts

- Why programming?
- Built-in data types
- Type conversion

Type Conversion

Type Conversion

Type conversion. Convert value from one data type to another.

- Automatic / Implicit: no loss of precision (promotion); or with strings.
- Explicit: cast; or method.
- Casting (higher precedence than arithmetic)

expression	value	type
explicit		
`str(2.718)`	`'2.718'`	`str`
`str(2)`	`'2'`	`str`
`int(2.718)`	`2`	`int`
`int(3.14159)`	`3`	`int`
`float(3)`	`3.0`	`float`
`int(round(2.718))`	`3`	`int`
implicit		
`3.0 * 2`	`6.0`	`float`
`10 / 4.0`	`2.5`	`float`
`math.sqrt(4)`	`2.0`	`float`

Typical type conversions

Cast

promotion

Method

String

expression	type	value
`'x: ' + str(99)`	`str`	`"x: 99"`
`2 * int(2.5)`	`int`	`4`
`math.sqrt(2)`	`float`	`2`
`11 + 0.25`	`float`	`11.25`
`(1+2+3+4)/4`	`float`	`2.5`

Precedence

Random Integer

Ex. Given N, generate a pseudo-random number between 0 and N-1.

Why not N?

```
import sys
import stdio
import random

def main():
    N = int(sys.argv[1])
    r = random.random()
    t = int(r * N)
    stdio.writeln(t)

if __name__ == '__main__': main()
```

str to int (explicit)

float to int (explicit)

int to float (implicit)

```
% python3 randomint.py 6
3

% python3 randomint.py 6
0

% python3 randomint.py 10000
3184
```

Summary

A data type is a set of values and a set of operations on those values.

Built-in data types in Python

- `str`, for computing with *sequence of characters*, for input and output.
- `int`, for computing with *integers*, for math calculations in programs.
- `float`, for computing with *floating point numbers*, typically for science and math apps.
- `bool`, for computing with *True* and *False*, for decision making in programs.

In Python you must:

- Declare and assign values to variables.
- Convert from one type to another when necessary.

Pay attention to the type of your data (even though a "dynamically-typed" language).

Python also has an interactive mode, useful as a calculator, has a help() function.