Ceng 111 – Fall 2018
Week 7

Dive into Python

Credit: Some slides are from the “Invitation to Computer Science” book by G. M. Schneider, J. L. Gersting and some from the “Digital Design” book by M. M. Mano and M. D. Ciletti.

Today

- Meet Python
- Basic & Structured Data
- Strings, tuples, lists, sets, dictionaries
- Expressions and their evaluation

Administrative Notes

- Live sessions schedule change
  - Tue 13:40: Common session
  - Wed 10:40: Section 3
  - ~~Wed 15:40: Section 2~~
  - ~~Thu 15:40: Section 1~~
- Social session
- The labs
- Midterm: 11 December 17:40
- Consent form

MEET python™

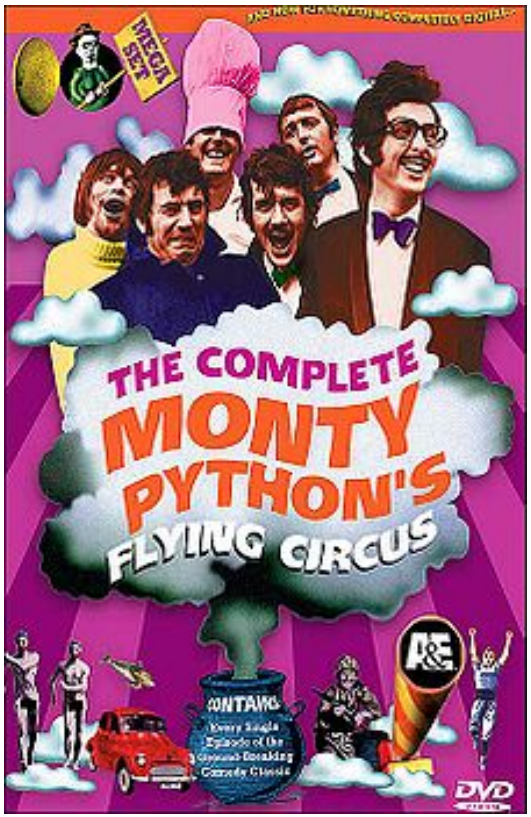
Guido van Rossum (1956 -)

- Zen of Python [https://en.wikipedia.org/wiki/Zen_of_Python]
  - Beautiful is better than ugly.
  - Explicit is better than implicit.
  - Simple is better than complex.
  - Complex is better than complicated.
  - Flat is better than nested.
  - Sparse is better than dense.
  - Readability counts.
  - …

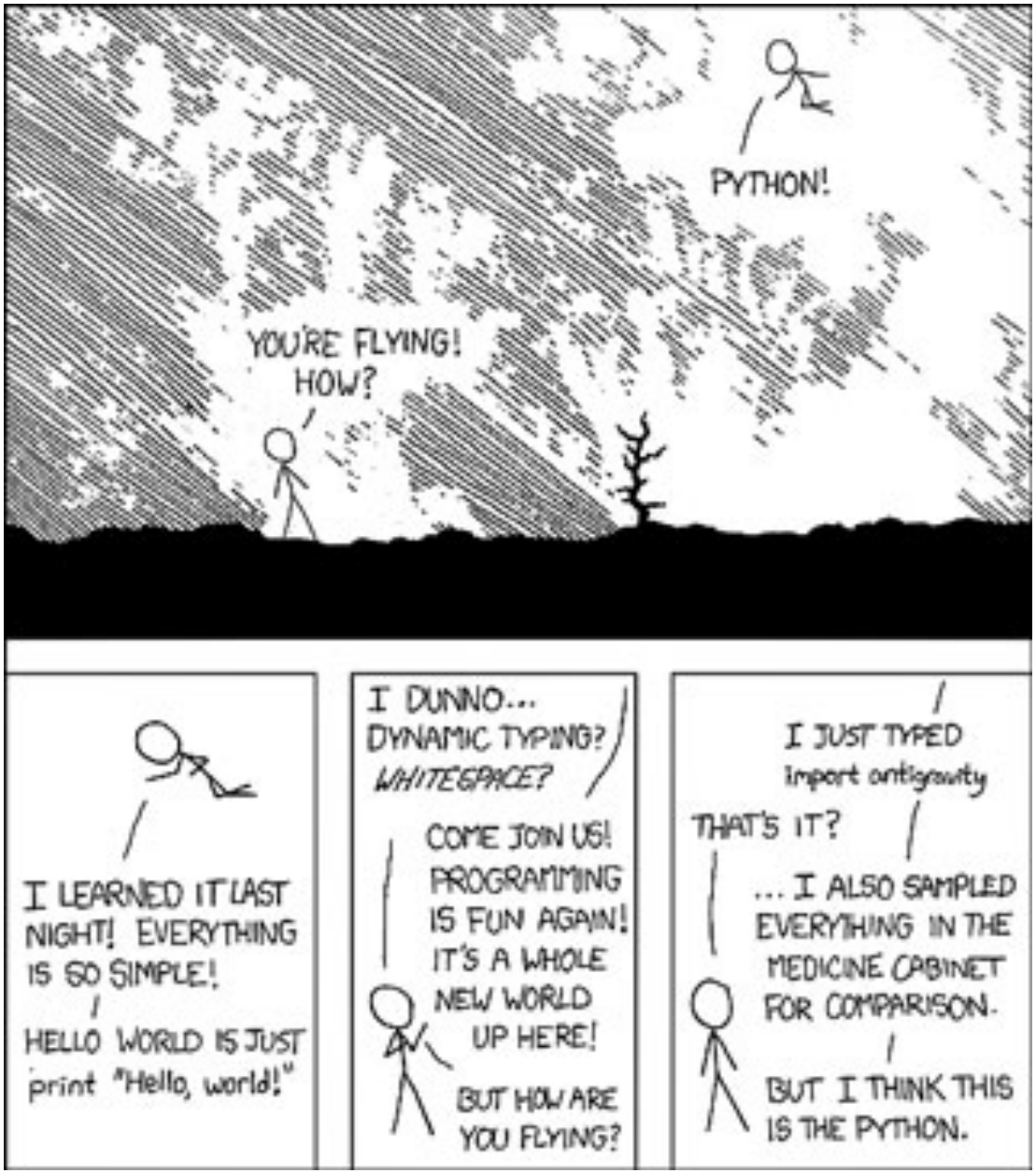
- An interpretive/scripting PL that:
  - Longs for code readability
  - Ease of use, clear syntax
  - Wide range of applications, libraries, tools
- Multiple Paradigms:
  - Functional
  - Imperative
  - Object-oriented

- Started at the end of 1980s.
- V2.0 was released in 2000
  - With a big change in development perspective: Community-based
  - Major changes in the facilities.
- V3.0 was released in 2008
  - Backward-incompatible
  - Some of its features are put into v2.6 and v2.7.

- Where does the name come from?
  - While van Possum was developing Python, he read the scripts of Monty Python's Flying Circus and thought 'python' was "short, unique and mysterious" for the new language [1]
- One goal of Python: "fun to use"
  - The origin of the name is the comedy group "Monty Python"
  - This is reflected in sample codes that are written in Python by the original developers.



[1] https://docs.python.org/2/faq/general.html#why-is-it-called-python

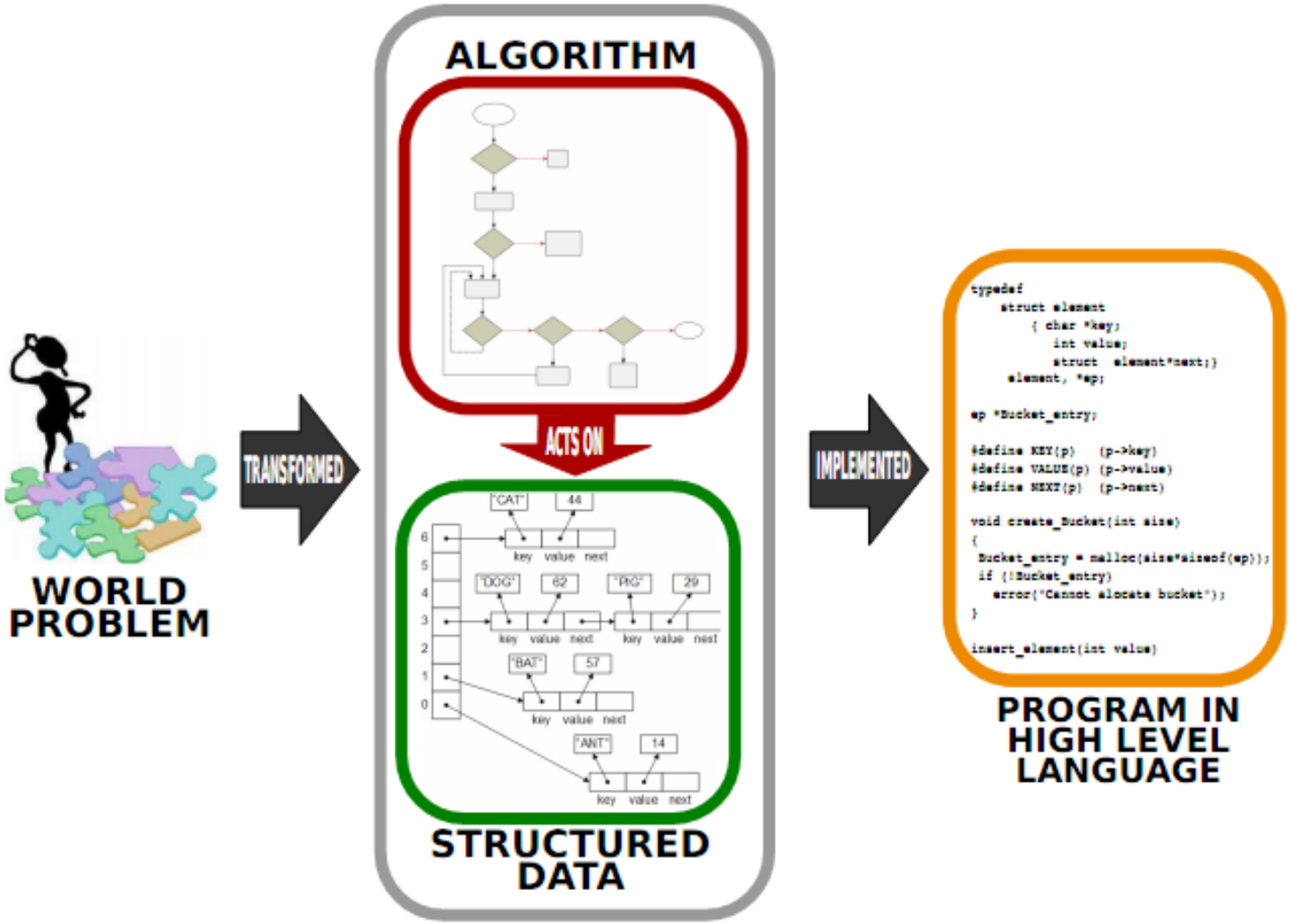

PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.


```
skalkan@divan:~$ python
Python 2.5.2 (r252:60911, Jan 24 2010, 17:44:40)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now

- Basic data & structured data

Design of a solution



What is data?

- **Data**: Information to be processed to solve a problem.
- Identify the data for the following example problems:
  - Find all wheat growing areas in a terrestrial satellite image.
  - Given the homework, lab and examination grades of a class, calculate the letter grades.
  - Alter the amplitude of a sound recording for various frequencies.
  - Extrapolate China's population for the year 2040 based on the change in the population growth rate up to this time.
  - Compute the launch date and the trajectory for a space probe so that it will pass by the outermost planets in the closest proximity.
  - Compute the layout of the internals of a CPU so that the total wiring distance is minimized.
  - Find the cheapest flight plan from A to B, for given intervals for arrival and departure dates.
  - Simulate a war between two land forces, given (i) the attack and the defense plans, (ii) the inventories and (iii) other attributes of both forces.

What is data?

- CPU can only understand two types of data:
  - Integers,
  - Floating points.
- The following are not directly understandable by a CPU:
  - Characters (‘a’, ‘A’, ‘2’, …)
  - Strings (“apple”, “banana”, …)
  - Complex Numbers
  - Matrices
  - Vectors
- But, programming languages can implement these data types.

Basic Data Types

- Integers
  - Full support from the CPU
  - Fast processing
- Floating Points
  - Support from the CPU with some precision loss
  - Slower compared to integer processing due to “interpretation”

Problems due to precision loss

- What has precision loss is misleading: 0.9 has precision loss whereas 0.9375 does not!
  - (0.9375 = 0.5 + 0.25 + 0.125 + 0.0625)
  - (½ + ¼ + 1/8 + 1/16) = 0.1111
  - Imprecise representations are round-offs!
  - If you have a lot of round-offs, you might be in trouble!

Problems due to precision loss

- 1.0023 – 1.0567
  - Result: -0.05440000000000004
- 1000.0023 – 1000.0567
  - Result: -0.05439999999986903
- Why?
  - Since the floating point representation is based on shifting the bits in the mantissa, the following are not equivalent in a PC
- π = 3.1415926535897931….
  - sin(π) should be zero
  - But it is not: $1.2246467991473532 \times 10^{-16}$

Problems due to precision loss

- Associativity in mathematics:
  - $(a+b) + c = a + (b + c)$
  - This does not hold in floating-point arithmetic:

set $a = 1234.567$, $b = 45.67834$ and $c = 0.0004$:
$(a + b) + c$ will produce 1280.2457399999998,
$a + (b + c)$ will produce 1280.2457400000001.

So what to do with precision loss, then?

- If you can transform the problem to the integer domain, do so.
  - Refrain from using floating points as much as you can.
- Use the most precise type of floating point of the high level language you are using.
  - Use only less precision floating points if you are short in memory.

So what to do with precision loss, then?

- If you have two numbers that are magnitude-wise incomparable, you are likely to lose the contribution of the smaller one. That will yield unexpected results when you repeat the addition in a computational loop where the looping is so much that the accumulation of the smaller is expected to become significant. It will not.
- The contrary happens too. Slight inaccuracies accumulate in loops to significant magnitudes and yield non-sense values.
- You better use well known, decent floating point libraries instead of coding floating point algorithms by yourself.

Integers in Python

- Python provides `int` type.

```
>>> type(3)
<type 'int'>
>>> type(3+4)
<type 'int'>
>>>
```

- For big integers, Python has the `long` type:

```
>>> type(3L)
<type 'long'>
>>> type(3L+4L)
<type 'long'>
>>>
```

`int` is limited by the hardware whereas `long` type is unlimited in Python.

Floating Points in Python

- Python provides `float` type.

```
>>> type(3.4)
<type 'float'>
>>>
```

```
>>> 3.4+4.3
7.7
>>> 3.4 / 4.3
0.7906976744186046
```

Simple Operations with Numerical Values in Python

Operator	Operator Type	Description
+	Binary	Addition of two operands
-	Binary	Subtraction of two operands
-	Unary	Negated value of the operand
*	Binary	Multiplication of two operands
/	Binary	Division of two operands
**	Binary	Exponentiation of two operands (Ex: x**y = x^y)

- **abs**(x): Absolute value of x.
- **round**(Float): Rounded value of float.
- **int**(Number), **long**(Number), **float**(Number): Conversion between numerical values of different types.

Characters

- We need characters to represent textual data:
  - Characters: ‘A’, …, ‘Z’, ‘0’, .., ‘9’, ‘a’, …, ‘z’ … etc.
- Computer uses the ASCII table for converting characters to binary numbers:

SYN	00010110	,	00101100	B	01000010	X	01011000	n	01101110
ETB	00010111	-	00101101	C	01000011	Y	01011001	o	01101111
CAN	00011000	.	00101110	D	01000100	Z	01011010	p	01110000
EM	00011001	/	00101111	E	01000101	(	01011011	q	01110001
SUB	00011010	0	00110000	F	01000110	\	01011100	r	01110010
ESC	00011011	1	00110001	G	01000111	)	01011101	s	01110011
FS	00011100	2	00110010	H	01001000	^	01011110	t	01110100
GS	00011101	3	00110011	I	01001001	_	01011111	u	01110101
RS	00011110	4	00110100	J	01001010	‘	01100000	v	01110110
US	00011111	5	00110101	K	01001011	a	01100001	w	01110111
SPC	00100000	6	00110110	L	01001100	b	01100010	x	01111000
!	00100001	7	00110111	M	01001101	c	01100011	y	01111001
"	00100010	8	00111000	N	01001110	d	01100100	z	01111010
#	00100011	9	00111001	O	01001111	e	01100101	}	01111011
$	00100100	:	00111010	P	01010000	f	01100110	|	01111100
%	00100101	;	00111011	Q	01010001	g	01100111	}	01111101
&	00100110	<	00111100	R	01010010	h	01101000	~	01111110
’	00100111	=	00111101	S	01010011	i	01101001	DEL	01111111
(	00101000	>	00111110	T	01010100	j	01101010		
)	00101001	?	00111111	U	01010101	k	01101011		
*	00101010	@	01000000	V	01010110	l	01101100		
+	00101011	A	01000001	W	01010111	m	01101101		

A portion of the ASCII table.

Characters in Python

- Python does not have a separate data type for characters!
- However, one character strings can be treated like characters:
  - **ord**(One_Char_String) : returns the ASCII value of the character in One_Char_String.
    - Ex: ord("A") returns 65.
  - **chr**(ASCII_value) : returns the character in a string that has the ASCII value ASCII_value:
    - Ex: chr(66) returns "B"

Boolean Values

- The CPU often needs to compare numbers, or data:
  - $3 >^{?} 4$
  - $125 =^{?} 1000/8$
  - $3 \leq^{?} 12345.34545/12324356.0$
- We have the truth values for representing the answers to such comparisons:
  - If correct: TRUE, True, true, T, 1
  - If not correct: FALSE, False, false, F, 0

Boolean Values in Python

- Python provides the `bool` data type for boolean values.
- `bool` data type can take `True` or `False` as values.

```
>>> 3 > 4
False
>>> type(3 > 4)
<type 'bool'>
```

```
>>> True == 2
False
>>> True == 1
True
>>> False == 0
True
>>> False == 1
False
```

Try this:

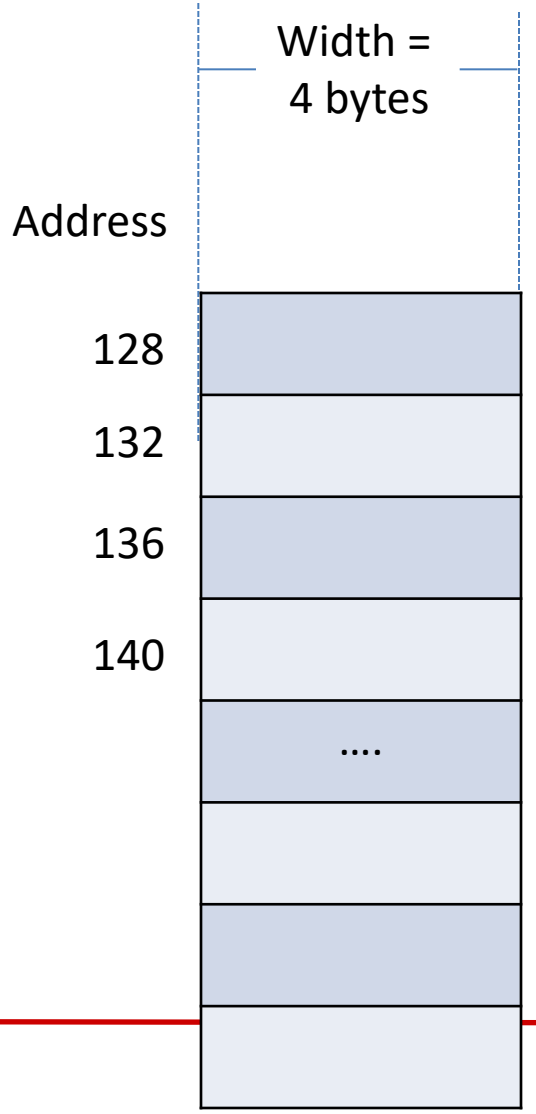
```
>>> True == bool(2)
```

not operator:

```
not True
not 4 > 3
```

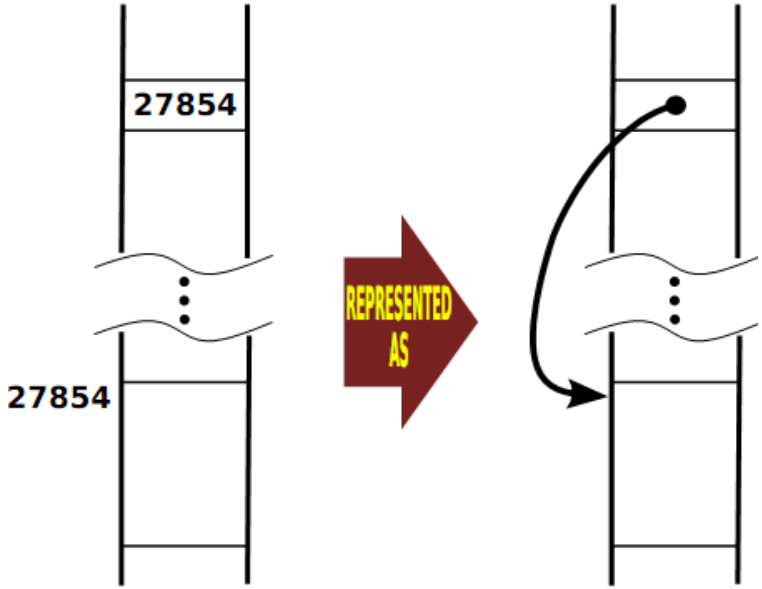
Structured Data

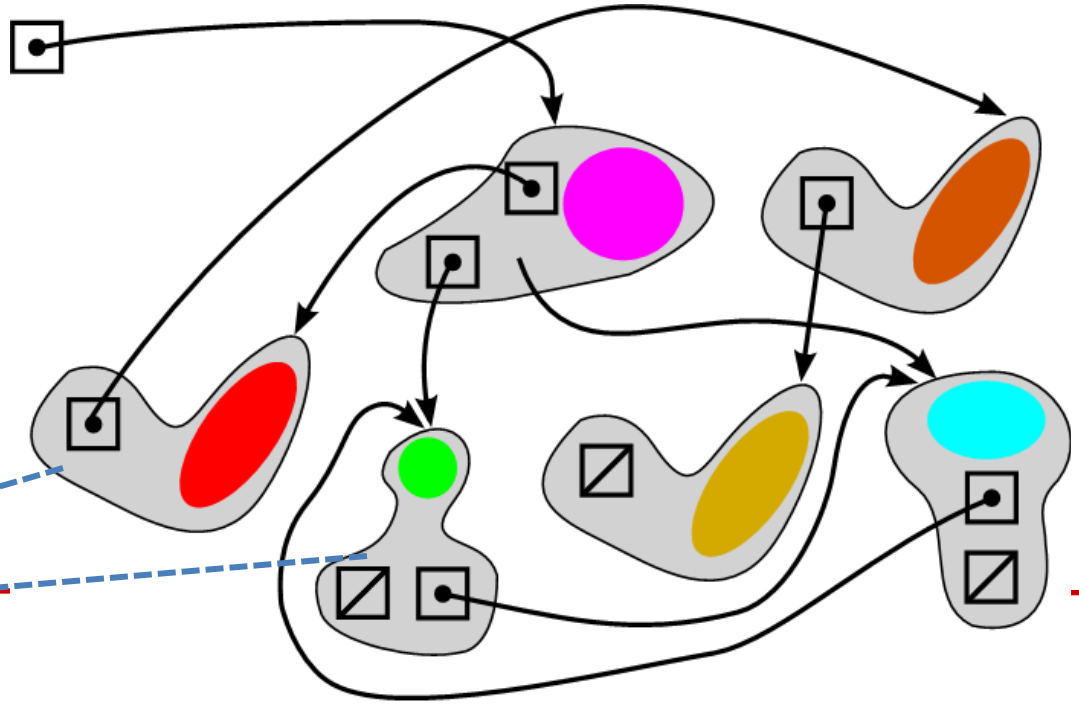
- If you have lots and lots of one type of data (for example, the ages of all the people in Turkey):
  - You can store them into memory consecutively (supported by most PLs)
    - This is called *arrays*.
  - Easy to access an element. Nth element:
    - <Starting-address>+ (N-1)*<Word Width>
    - Ex: 2nd element is at 128 + (2-1) * 4 = 132



Structured Data

- What if you have to make a lot of deletions and insertions in the middle of an array?
- Then, you have to store your data in blocks/units such that each unit has the starting address of the next unit/block.





Strings

- Sequence of characters:
  - Ex: “Book”, “Programming”, “Python”
- How can they be represented?
  1. Put a set of characters one after the other and end them with a non-character value.
  2. At the beginning of the characters, specify how many characters follow.
- Both have advantages and disadvantages.

Strings in Python

Python provides the **str** data type for strings:

```
>>> "Hello?"
'Hello?'
>>> type("Hello?")
<type 'str'>
```

- Simplest operation with a string:

```
>>> len("Hello?")
6
```

Strings in Python

- Accessing elements of a string

  “Hello?”[0] → 1st character (i.e., “H”)

  “Hello?”[4] → 5th character

- Indexing starts at 0!!!
- What is the last element then?
  - “Hello?”[len(“Hello?”) - 1]
- Negative indexing possible:
  - Last element: “Hello?”[-1] → “?”
- In general:
  - String[start:end:step]
  - Ex: “Hello?”[0:4:2] → “Hl”
  - Ex: “Hello?”[2:4] → “ll”

Creating Strings in Python

1. Enclosing a set of characters between quotes:
   - “ali”, “veli”, “deli”, …
2. Using the str() function:
   - str(4.5) → “4.5”
3. Using the raw_input() function:

```
>>> a = raw_input("--> ")
--> Do as I say
>>> a
'Do as I say'
>>> type(a)
<type 'str'>
```

Internal of Python's String Implementation

(taken from https://www.laurentluce.com/posts/python-string-objects-implementation/)

- PyStringObject structure
  - A string object in Python is represented internally by the structure PyStringObject. "ob_shash" is the hash of the string if calculated. "ob_sval" contains the string of size "ob_size". The string is null terminated. The initial size of "ob_sval" is 1 byte and ob_sval[0] = 0. If you are wondering where "ob_size is defined", take a look at PyObject_VAR_HEAD in object.h.

```
typedef struct {
    PyObject_VAR_HEAD
    long ob_shash;
    int ob_sstate;
    char ob_sval[1];
} PyStringObject;
```

Tuples

- Tuple: ordered set of data:
  - (1, 2, 3)
  - (“a”, “b”, “c”)
- May be heterogeneous:
  - (“Salary”, 2000, “Age”, 25, “Birth”, “Ankara”)

Tuples in Python

```
>>> (1, 2, 3, 4, "a")
(1, 2, 3, 4, 'a')
>>> type((1, 2, 3, 4, "a"))
<type 'tuple'>
```

- Tuples in Python: collection of data enclosed in parentheses, separated by comma.
- Accessing elements of a tuple (like strings):
  - Positive Indexing: `(1, 2, 3, 4, "a")[2]` returns 3.
  - Negative Indexing: `(1, 2, 3, 4, "a")[-1]` returns `'a'`.
  - Ranged Indexing, *i.e.*, `[start:end:step]`: `(1, 2, 3, 4, "a")[0:4:2]` leads to `(1, 3)`.

Creating Tuples in Python

1. Enclosing data within parentheses:
   - Ex: (1, “a”, “cde”, 23)
2. Using the tuple() function:
   - Ex: tuple(“ABC”) → (‘A’, ‘B’, ‘C’)
3. Using the input() function:

```
>>> a = input("Give me a tuple:")
Give me a tuple:(1, 2, 3)
>>> a
(1, 2, 3)
>>> type(a)
<type 'tuple'>
```

Lists

- Similar to tuples.
- Difference:
  - Tuples are immutable (i.e., not changeable) whereas lists are mutable.

Lists in Python

```
>>> [1, 2, 3, 4, "a"]
[1, 2, 3, 4, 'a']
>>> type([1, 2, 3, 4, "a"])
<type 'list'>
```

- Lists in Python: collection of data enclosed in brackets, separated by comma.
- Accessing elements of a list (like strings & tuples):

- Positive Indexing: `[1, 2, 3, 4, "a"][2]` returns 3.
- Negative Indexing: `[1, 2, 3, 4, "a"][-1]` returns `'a'`.
- Ranged Indexing, *i.e.*, `[start:end:step]`: `[1, 2, 3, 4, "a"][0:4:2]` leads to `[1, 3]`.

Creating Lists in Python

1. Enclosing data within brackets:
   - Ex: [1, “a”, “cde”, 23]
2. Using the list() function:
   - Ex: list(“ABC”) → [‘A’, ‘B’, ‘C’]
3. Using the range() function:

```
range(   [start,] stop[, step])
```

   - Ex: range(1, 10, 2) → [1, 3, 5, 7, 9]
4. Using the input() function:

```
>>> a = input("Give me a list:")
Give me a list:[1, 2, "a"]
>>> a
[1, 2, 'a']
>>> type(a)
<type 'list'>
```

Modifying a List in Python

- List[range] = Data
  - Ex:

```
>>> L = [3, 4, 5, 6, 7, '8', 9, '10']
>>> L[::2]
[3, 5, 7, 9]
>>> L[::2] = [4, 6, 8, 10]
>>> L[::2]
[4, 6, 8, 10]
>>> L[]
[4, 4, 6, 6, 8, '8', 10, '10']
```

- Using the append() function:
  - List.append(item)
  - Ex: [1, 2, 3].append(5) → [1, 2, 3, 5]

Modifying a List in Python

- Using the extend() function:
  - List.extend(Another_list)
  - Ex: [1, 2, 3].extend([“a”, “b”]) → [1, 2, 3, “a”, “b”]

```
>>> L.extend(["a", "b"])
>>> L
[4, 4, 6, 6, 8, '8', 10, '10', 'a', 'a', 'b']
```

- Using the insert() function:
  - List.insert(index, item)

```
>>> L=[1, 2, 3]
>>> L
[1, 2, 3]
>>> L.insert(1, 0)
>>> L
[1, 0, 2, 3]
```

Removing Elements from a List in Python

- **del statement:** `del L[start:end]`

```
>>> L
[1, 0, 2, 3]
>>> del L[1]
>>> L
[1, 2, 3]
```

- **L.pop() function:** `L.pop([index])`

```
>>> L=[1,2,3]
>>> L.pop()
3
>>> L
[1, 2]
>>> L.pop(0)
1
>>> L
[2]
```

- **L.remove() function:** `L.remove(value)`

```
>>> L
[2, 1, 3]
>>> L.remove(1)
>>> L
[2, 3]
```

A frequent operation with containers

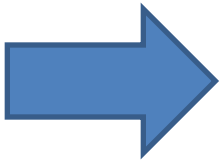
- Membership
  - in
  - not in
- “en” in “deneme”
  - True
- “an” in “deneme”
  - False
- “dem” in “deneme”
  - False

Accessing Data/Containers by Names: Variables

- Naming:
  - Usually: A combination of letters and numbers
  - Ex: a123, 123a, …
- Scope & Extent:
  - Scope: Where a variable can be accessed.
  - Extent: The lifetime of a variable.
- Typing:
  - Statically typed: The type of a variable is fixed.
  - Dynamically typed: The type of a variable is variable ☺

Variables in Python


```
>>> a = 4
>>> b = 3
>>> c = a + b
>>> a
4
>>> b
3
>>> c
7
```



- We don't need to define a variable before using it.
- We don't need to specify the type of a variable.

- '=' means "Change the content of the variable with the value at the right-hand side".
  - Assignment!
- The left-side of the assignment should be a valid variable name:
  - Ex: a+2 = 5 → NOT VALID!

Variable Naming in Python

- Variable names are case sensitive. So, the names `a` and `A` are two different variables.
- Variable names can contain letters from the English alphabet, numbers and an underscore `_`.
- Variable names can only start with a letter or an underscore. So, `10a`, `$a`, and `var$` are all invalid whereas `_a` and `a_20`, for example, are valid names in Python.

- Variable names cannot be one of the keywords in Python:

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

More on Variables in Python

- Typing of variables:
  - Python is dynamically typed:

```
>>> a = 3
>>> type(a)
<type 'int'>
>>> a = 3.4
>>> type(a)
<type 'float'>
```

- Using variables:

```
>>> a = (1, 2, 3, 'a')
>>> type(a)
<type 'tuple'>
>>> a[1]
2
>>> a[-1]
'a'
```

Variables, Values and Aliasing in Python

- Every data (whether constant or not) has an identifier (an integer) in Python:

```
>>> a = 1
>>> b = 1
>>> id(1)
135720760
>>> id(a)
135720760
>>> id(b)
135720760
```

- If the type of the data is mutable, there is a problem!!!

```
>>> a = ['a', 'b']
>>> b = a
>>> id(a)
3083374316L
>>> id(b)
3083374316L
>>> b[0] = 0
>>> a
[0, 'b']
```

This is called Aliasing.

```
a = 4
b = [1,2,3,a]
a = 8
print b
```

```
>>> a=[1,2]
>>> b=[1,2,a]
>>> a
[1, 2]
>>>
>>> b
[1, 2, [1, 2]]
>>> a.append(3)
>>> b
[1, 2, [1, 2, 3]]
>>> a
[1, 2, 3]
```

- When you use basic data, it is evaluated.
- When you use container data, its address (pointer) is passed.

How to make copy of the list?

- list(List-to-be-copied)
- L[:]
- Shallow copy

  import copy

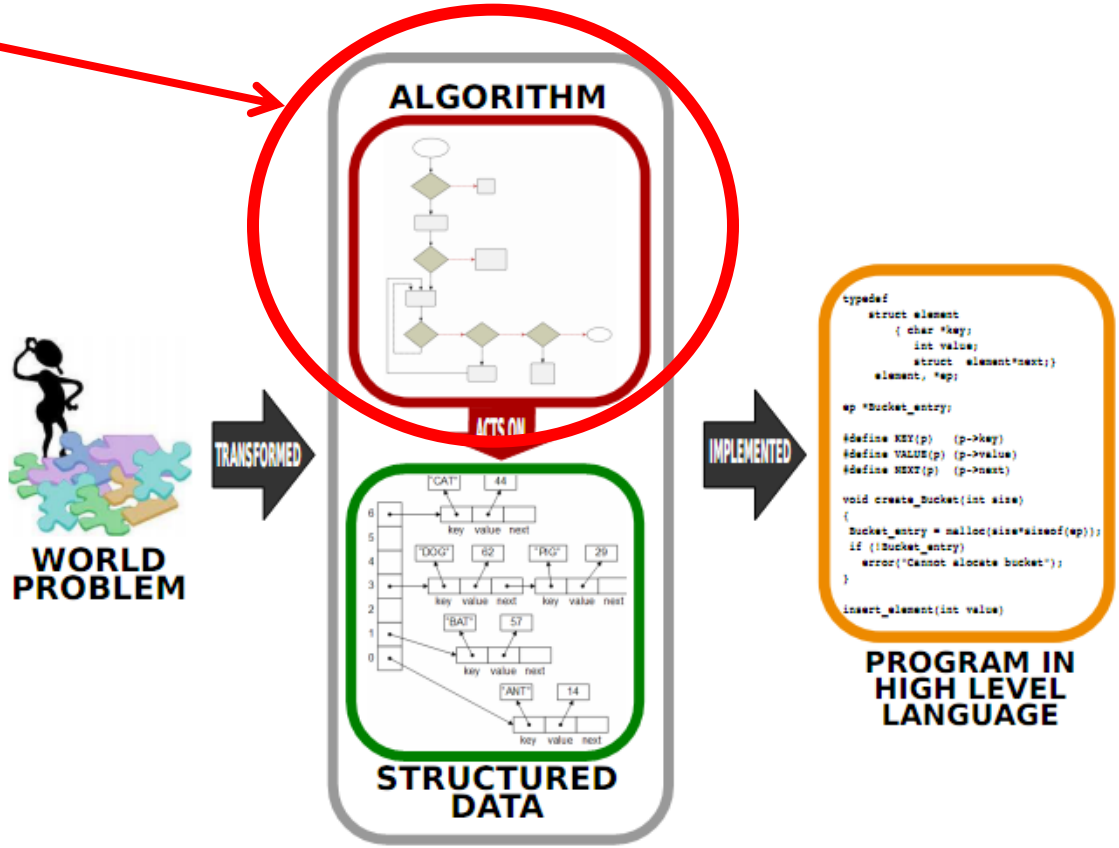
  copy.copy(list)
- Deep copy

  import copy

  copy.deepcopy(list)

Now

- We start another ingredient of a program:
  - Actions!



What are actions?

- Actions in a PL are the *things* that we can do with the data. What could they be?
  - Create data or modify data
  - Interact with the external environment

Actions for creating/modifying data

- Evaluating a mathematical expression
  - But there are differences to the expressions in Mathematics
- Working with structured data
- Storing results of computations (in another data)
- Making a decision about how to proceed with the computation
  - `if x*y < 3.1415 then` ⟨*do some action*⟩
  - `if "ali" in class_111_list then` ⟨*do some action*⟩
  - `if tall("ali") then` ⟨*do some action*⟩

Interaction-type actions

- “Interaction” means Input/Output.
- Why interact with the environment? Why do we have Input/Output actions?
  - To react on a change in the external environment
  - To produce an effect in the external environment

Action Types in High-Level Languages

- Expression evaluation
  - 3 + 4 * 5 / 2

vs.

- Statement execution
  - del L[2:4]

Expressions

- An expression is a calculation which has a set of *operations*.
- Operations have *operators* and *operands*.
- Example: 3 + 4
  - + → operator
  - 3, 4 → operands

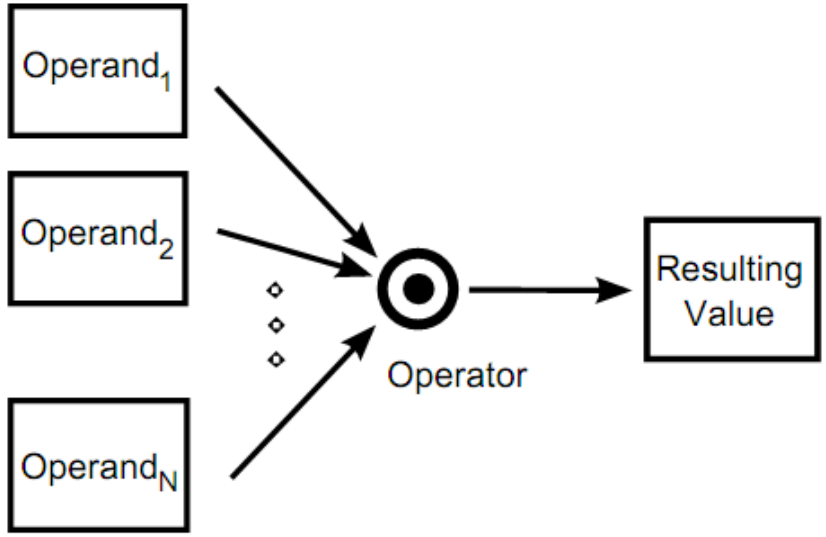
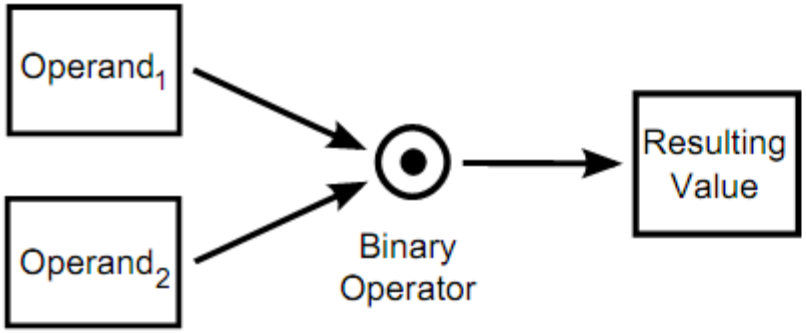
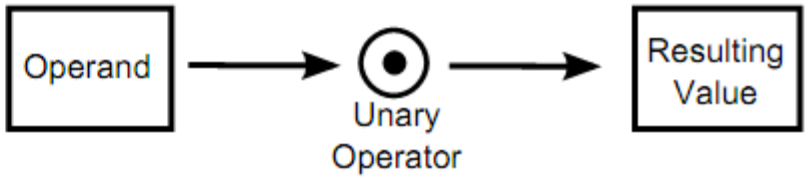


Figure 3.1: N-ary operation



(a) Binary operation



(b) Unary operation

Expressions in Python
- Involving arithmetic operators -

Operator	Operator Type	Description
+	Binary	Addition of two operands
-	Binary	Subtraction of two operands
-	Unary	Negated value of the operand
+	Unary	Positive value of the operand
*	Binary	Multiplication of two operands
/	Binary	Division of two operands
**	Binary	Exponentiation of two operands (Ex: `x**y` $= x^y$)

Expressions in Python
- Involving arithmetic operators -

- Precedence & Associativity of arithmetic operators.
- What is precedence?
  - The expression “3 + 4 * 5” has two different interpretations:
    - (3+4)*5
    - 3 + (4*5)
- What is associativity?
  - The expression “3.02 + 4.1 + 5.24” has two different interpretations:
    - (3.02+4.1)+5.24
    - 3.02+(4.1+5.24)

Expressions in Python
- Involving arithmetic operators -

- Precedence & Associativity of arithmetic operators.
- Top: highest precedence.
- Bottom: lowest precedence.

Operator	Type	Associativity	Description
**	Binary	Right-to-left	Exponentiation
+, -	Unary	Right-to-left	Positive, negative
*, /, //, %	Binary	Left-to-right	Multiplication, Division, Remainder, Modulo
+, -	Binary	Left-to-right	Addition, Subtraction

Floor division
(fraction part of the result is removed)

Expressions in Python
- Involving container operators -

- Concatenation (+)
  - “a” + “b” → “ab”
- Repetition (*)
  - “a” * 3 → “aaa”
- Membership (in, not in):
  - “a” in “Mathematics” → True
  - “a” not in “Mathematics” → False
- Indexing ([])

Expressions in Python
- Involving container operators -

- Precedence and associativity of container operators

Operator	Type	Associativity	Description
[]	Binary	Left-to-right	**Indexing**
**	Binary	Right-to-left	Exponentiation
+, -	Unary	Right-to-left	Positive, negative
*, /, //, %	Binary	Left-to-right	Multiplication & **Repetition**, Division, Remainder, Modulo
+, -	Binary	Left-to-right	Addition, Subtraction, **Concatenation**
in, not in	Binary	Right-to-left	Membership

Expressions in Python
- Involving relational operators -

- Equality (==)
  - Two data are equivalent if they represent the same value/information!
  - "Ali" == "Ali" → True
- Less-than (<):
  - A numerical data is less than another if the value of the first is less than that of the second:
    - 3 < 4.5 → True
  - A string is less than another if it is lexicographically (i.e., in ASCII value) less than the second.
    - "abc" < "def" → True
  - A tuple/list is less than another tuple/list if the first different items satisfy the less-than relation.

Expressions in Python
- Involving relational operators -

- Less-than-or-equal (<=)
  - <= → (<) or (==)
- Greater-than (>)
  - > → not (<=)
- Greater-than-or-equal-to (>=)
  - >= → not (<)
- Not-equal (!=)
  - != → not (==)

Note that in Python, relational operators can be chained. In other words, `a RO b RO c` (where `RO` is a relational operator) is interpreted as:

`(a RO b) and (b RO c)`.

In most other programming languages, `a RO b RO c` is interpreted as `(a RO b) RO c`.

Expressions in Python
- Involving relational operators -

- Precedence & Associativity

Operator	Type	Associativity	Description
[]	Binary	Left-to-right	**Indexing**
**	Binary	Right-to-left	Exponentiation
+, -	Unary	Right-to-left	Positive, negative
*, /, //, %	Binary	Left-to-right	Multiplication & Repetition, Division, Remainder, Modulo
+, -	Binary	Left-to-right	Addition, Subtraction, Concatenation
in, not in, <, <= >, >=, ==, !=	Binary	Right-to-left	Membership, **Comparison**

Expressions in Python
- Involving logical operators -

- Logical operators manipulate truth values:
- and operator
  - A and B → True iff (A is True) & (B is True)
- or operator
  - A or B → True iff either (A is True) or (B is True)
- not operator
  - not A → True iff A is False

Expressions in Python
- Involving logical operators -

- Precedence & Associativity

Operator	Type	Associativity	Description
[]	Binary	Left-to-right	**Indexing**
**	Binary	Right-to-left	Exponentiation
+, -	Unary	Right-to-left	Positive, negative
*, /, //, %	Binary	Left-to-right	Multiplication & Repetition, Division, Remainder, Modulo
+, -	Binary	Left-to-right	Addition, Subtraction, Concatenation
in, not in, <, <= >, >=, ==, !=	Binary	Right-to-left	Membership, Comparison
not	Unary	Right-to-left	Logical negation
and	Binary	Left-to-right	Logical AND
or	Binary	Left-to-right	Logical OR

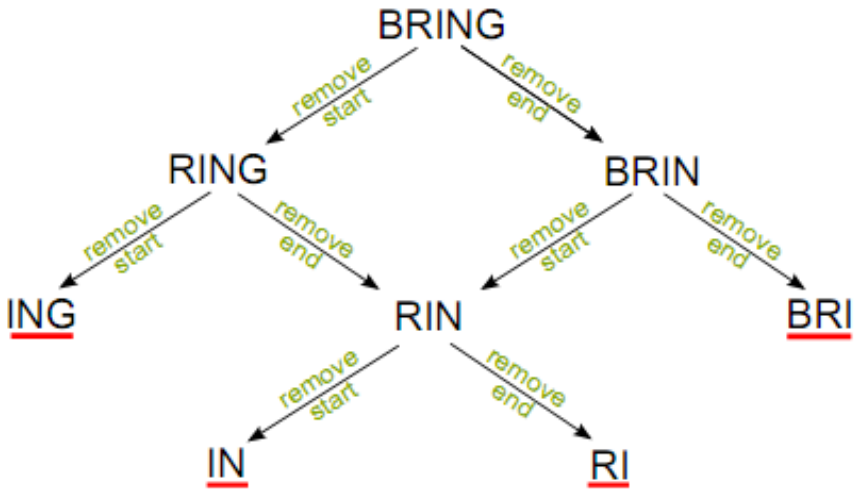
Expressions in Python
- assignment (**not** an operator) -

- Single assignment:
  - a = 4
- Multiple assignment:
  - a = b = c = 4
- Combined assignment:
  - a = a + 4 → a += 4
  - +=, *=, -=, /=, etc.

```
>>> b += 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>> b = 5
>>> b **= 2
>>> b
25
```

Church-Rosser Property

- A reduction/re-writing system has the Church-Rosser Property if the set of rules always lead to the same results independent of the order of application of the rules.
- Evaluation of a mathematical expression is said to have the Church-Rosser Property:
- A simple example:
  - “If both ends of a string are consonants, remove one”



Church-Rosser Property

- How about expressions in programming languages? Do they have Church-Rosser Property?
- Answer it yourself considering these:
  - Limitations due to fixed size representations of numbers: Remember that a+(b+c) may not be equivalent to (a+b)+c?
  - Side-effects in evaluating some operations and function calls
    - f(2) + x

LESSON: A programmer has to know the order an expression is evaluated!

So, how are expressions evaluated in HLPL?

- Consider these:
  - `2 - 3 ** 4 / 8 + 2 * 4 ** 5 * 1 ** 8`
  - `4 + 2 - 10 / 2 * 4 ** 2`
  - `3 / 3 ** 3 * 3`
- or these:

a) `not a == b + d < not a`

b) `a == b <= c == True`

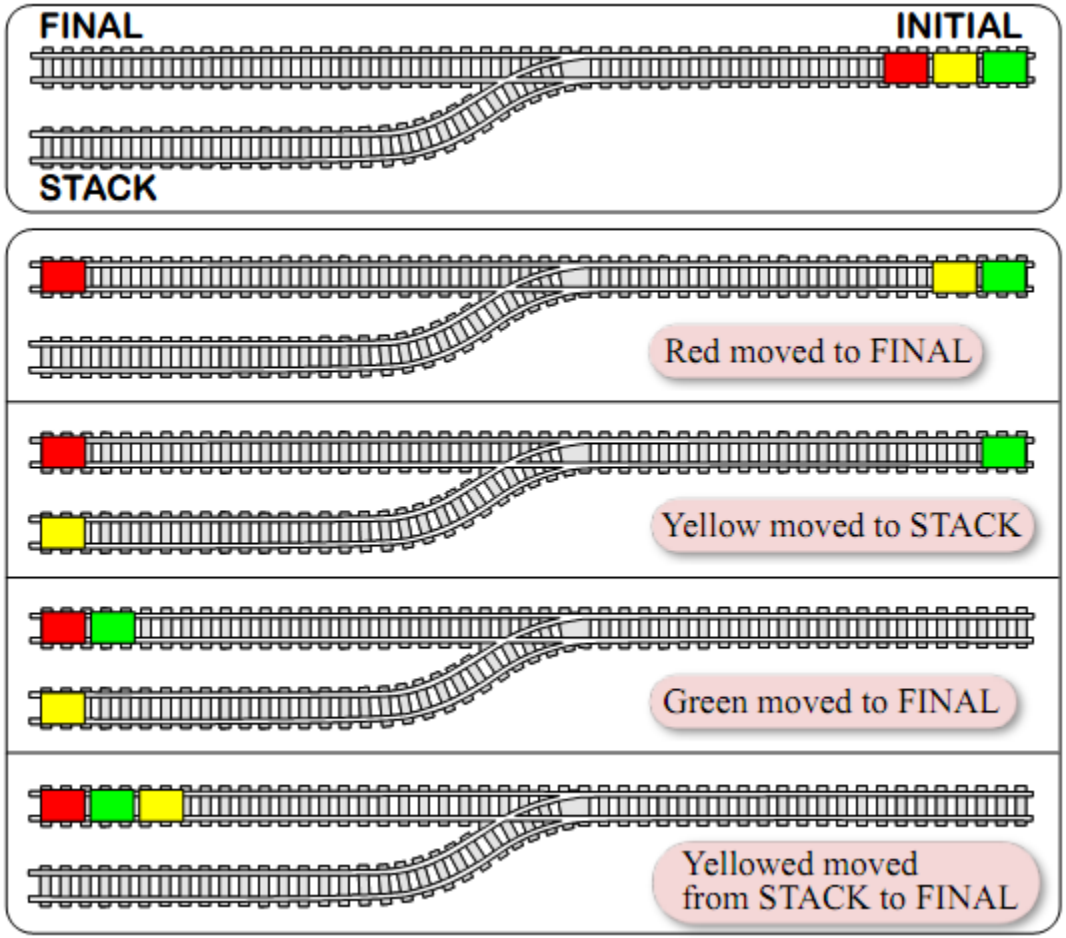
c) `True <= False == b + c`

d) `c / a / b`

Expression Evaluation in PLs

- The PL –most of them at least- do not define the order an expression is evaluated.
  - The compiler/interpreter does!
- In most cases,
  - First, *Dijkstra's shunting-yard algorithm* is used to convert an expression into postfix notation.
  - Then, the postfix expression is evaluated using a *postfix evaluation algorithm*.

Dijktsra's Shunting-Yard Algorithm

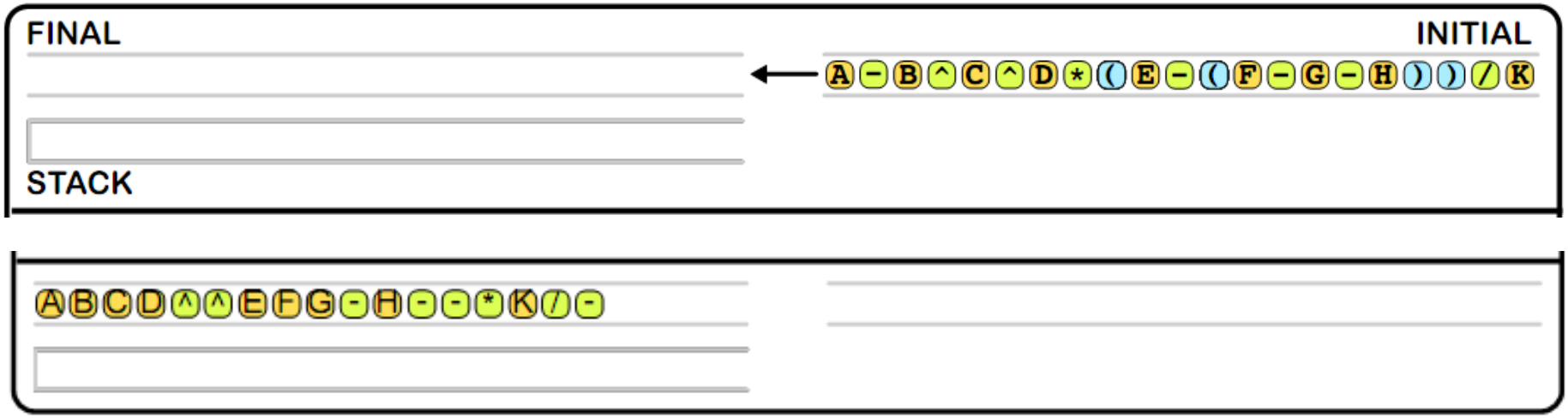


Algorithm 1 Dijkstra's Shunting-yard algorithm.

Get next token t from the input queue
if t is an operand **then**
  Add t to the output queue
if t is an operator **then**
  while There is an operator τ at the top of the stack, and either t is left-associative and its precedence is less than or equal to the precedence of τ, or t is right-associative and its precedence is less than the precedence of τ **do**
    Pop τ from the stack, to the output queue.
  Push t on the stack.
if t is a left parenthesis **then**
  Push t on the stack.
if t is a right parenthesis **then**
  Pop the operators from the stack, to the output queue until the top of the stack is a left parenthesis.
  Pop the left parenthesis.
if No more tokens to get **then**
  Pop the operators on the stack, if any, to the output queue.

Dijktsra's Shunting-Yard Algorithm: Example

$$A + \frac{B^{C^D} \times (E - (F - G - H))}{K}$$



Postfix Evaluation

1. Go from left to right
2. When you see an operator:
   a) Apply it to the last two operands
   b) Remove the last two operands and put the result in place of the operator.