CS46K Programming Languages

Structure and Interpretation of Computer Programs

Chapter 2 Data Type and Type Systems

LECTURE 2: MEMORY MODEL AND DATA TYPES

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Data Types
- Type System
- Primitive Data Types
- Name, Value and Expressions
- Composite Data Types

Data Types

SECTION 1

data types

Data is a collection of information which may include facts, numbers, measurements or other information.

Data is often organised in graphs or charts for statistical analysis. How this is done depends on what type of data it is.

Types of data

data

qualitative or categorical data
Data describing qualities, characteristics or categories.

quantitative or numerical data
Data where quantities can be counted or measured.

qualitative/categorical
Categorical variables cannot be used to calculate averages.

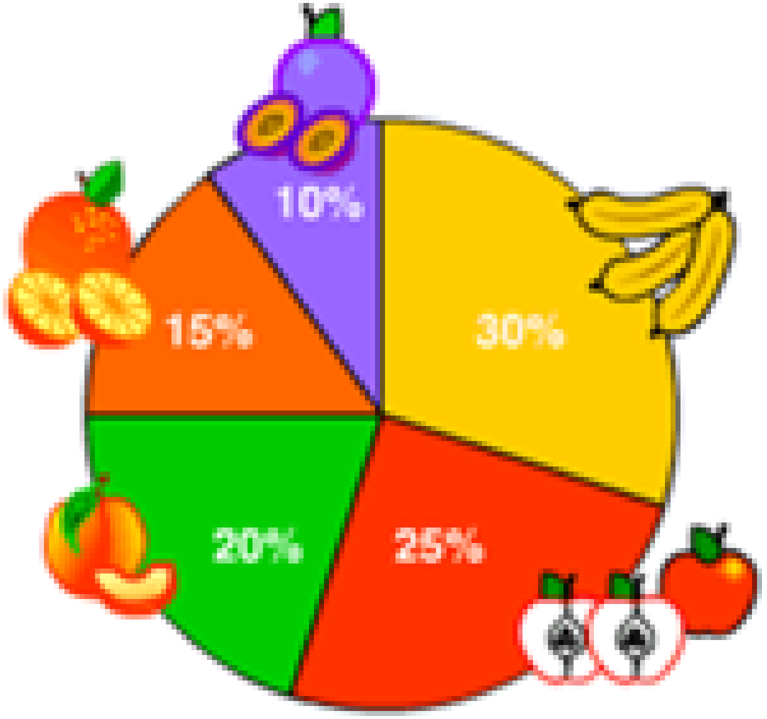
quantitative/numerical
Numerical variables can be used to calculate averages.

discrete
(finite values)
Data that is counted e.g. sales figures.

continuous
(infinite values)
Data that is measured e.g. temperature.

qualitative

Student Survey favourite fruits



quantitative

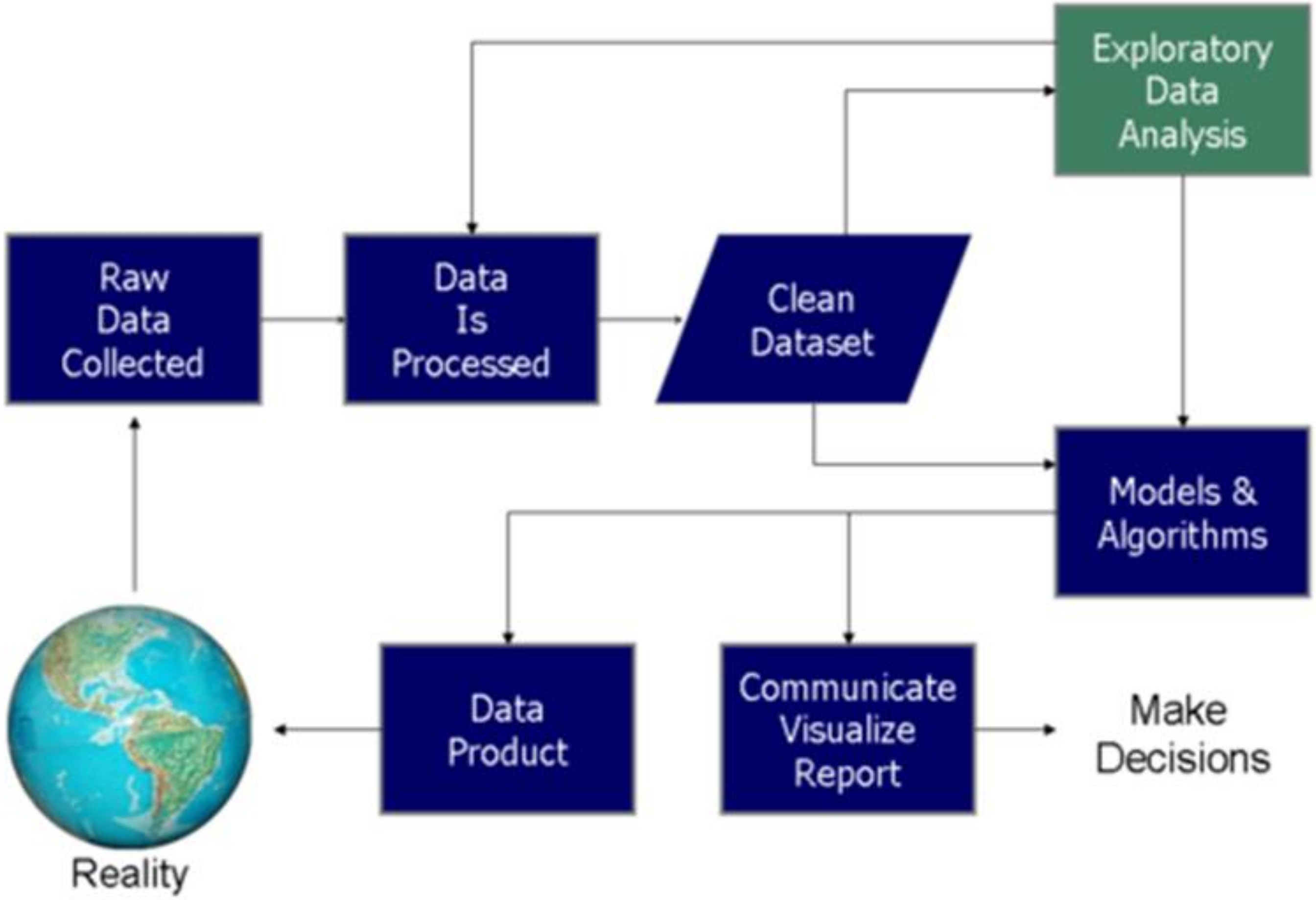
Daily Fruit Sales Numbers Convenience Store

20 + 60 + 80 + 100 + 50 = 310

310 ÷ 5 = 62

An average of 62 pieces of fruit per day sold.

Data Science Process



Data Types

- We all have developed an intuitive notion of what types are; what's behind the intuition?
  - collection of values from a "domain" (the denotational approach)
  - internal structure of a bunch of data, described down to the level of a small set of fundamental types (the structural approach)
  - collection of well-defined operations that can be applied to objects of that type (the abstraction approach)

Data Types

Purpose of Data Types

- **Associated with a Set of Operations**: types provide implicit context for many operations. In object-oriented environments, they are called methods.

- **Semantical Validity:** Types limit the set of operations that may be performed in a semantically valid program. They prevent the programmer from adding a character and a record, or from taking the arctangent of a set.

- **Higher Readability:** It types are specified explicitly in the source program, they can often make the program easier to read and understand.

- **Performance Optimization:** If types are known at compile time, they can be used to drive important performance optimizations.

Data Types

- What are types good for?
  - implicit context
  - checking - make sure that certain meaningless operations do not occur
    - type checking cannot prevent all meaningless operations
    - It catches enough of them to be useful
- Polymorphism results when the compiler finds that it doesn't need to know certain things

Type System

SECTION 2

Data Types

Dynamic Typing: determine data type at run-time.

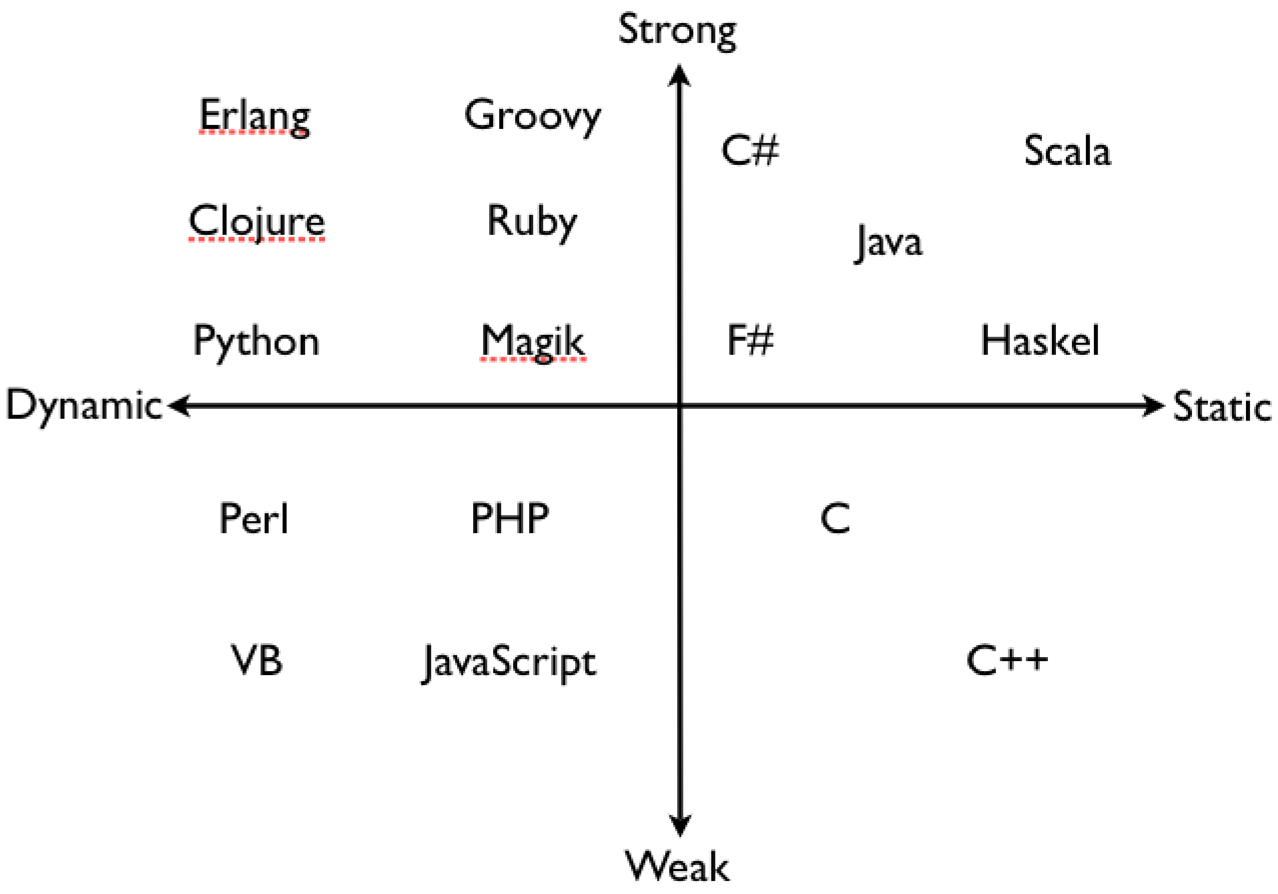
- The growing popularity of scripting languages has led a number of prominent software developers to publicly question the value of static typing.
- They ask: given that we can't check everything at compile time, how much pain is it worth to check the things we can? As a general rule, it is easier to write type-correct code than to prove that we have done so, and static typing requires such proofs. The complexity of static typing increases correspondingly.
- Anyone who has written extensively in Ada or C++ on the one hand, and in Python or Scheme on the other, cannot help but be struck at how much easier it is to write code.

Data Types

Dynamic Typing: determine data type at run-time

- Dynamic checking incurs some run-time overhead and may delay the discovery of bugs, but this is increasingly seen as insignificant in comparison to the potential increase in human productivity.
- The choice between static and dynamic typing promises to provide one of the most interesting language debates of the coming decade.

Languages



Strong
Erlang
Groovy
C#
Scala
Clojure
Ruby
Java
Python
Magik
F#
Haskel
Dynamic
Static
Perl
PHP
C
VB
JavaScript
C++
Weak

Type Systems

Examples

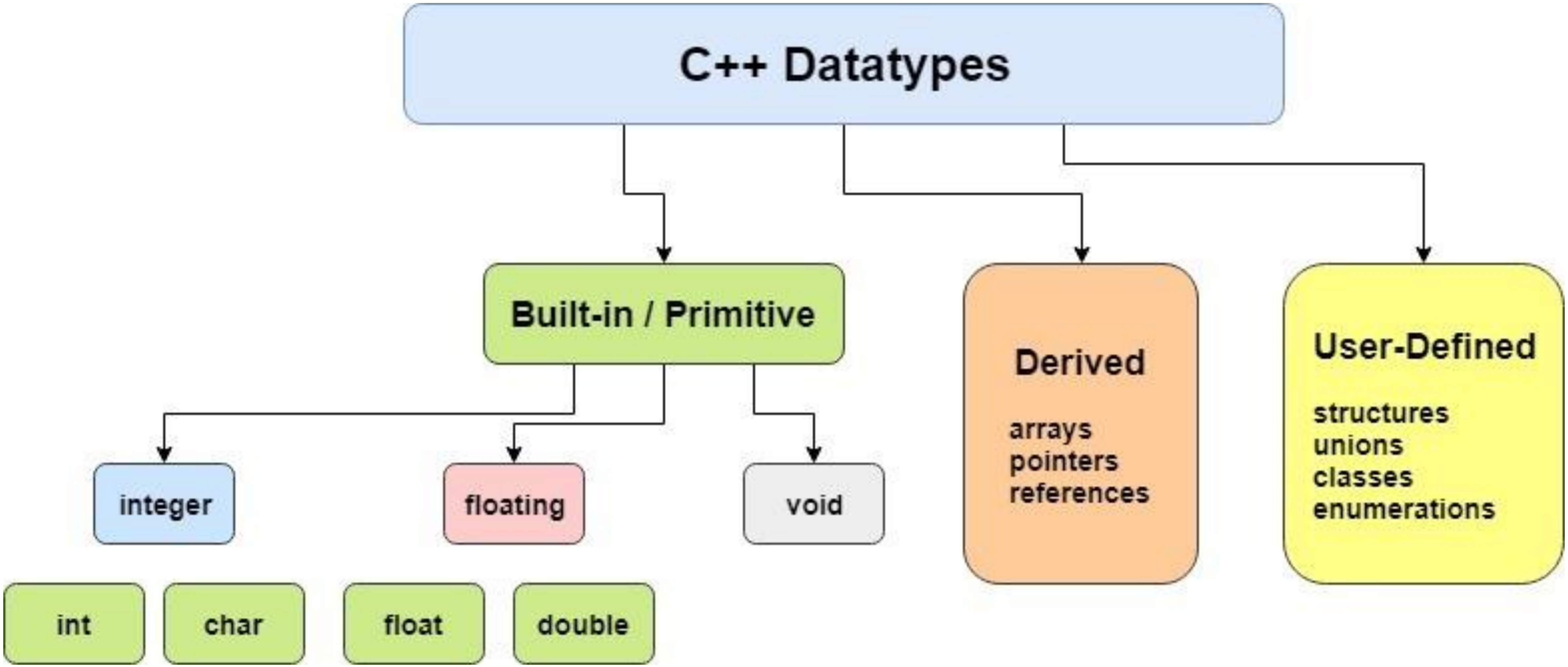
- Common Lisp is strongly typed, but not statically typed
- Ada is statically typed
- Pascal is almost statically typed
- Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically
- C has become more strongly typed with each new version, though loopholes still remain

Overview of Chapter on Data Types

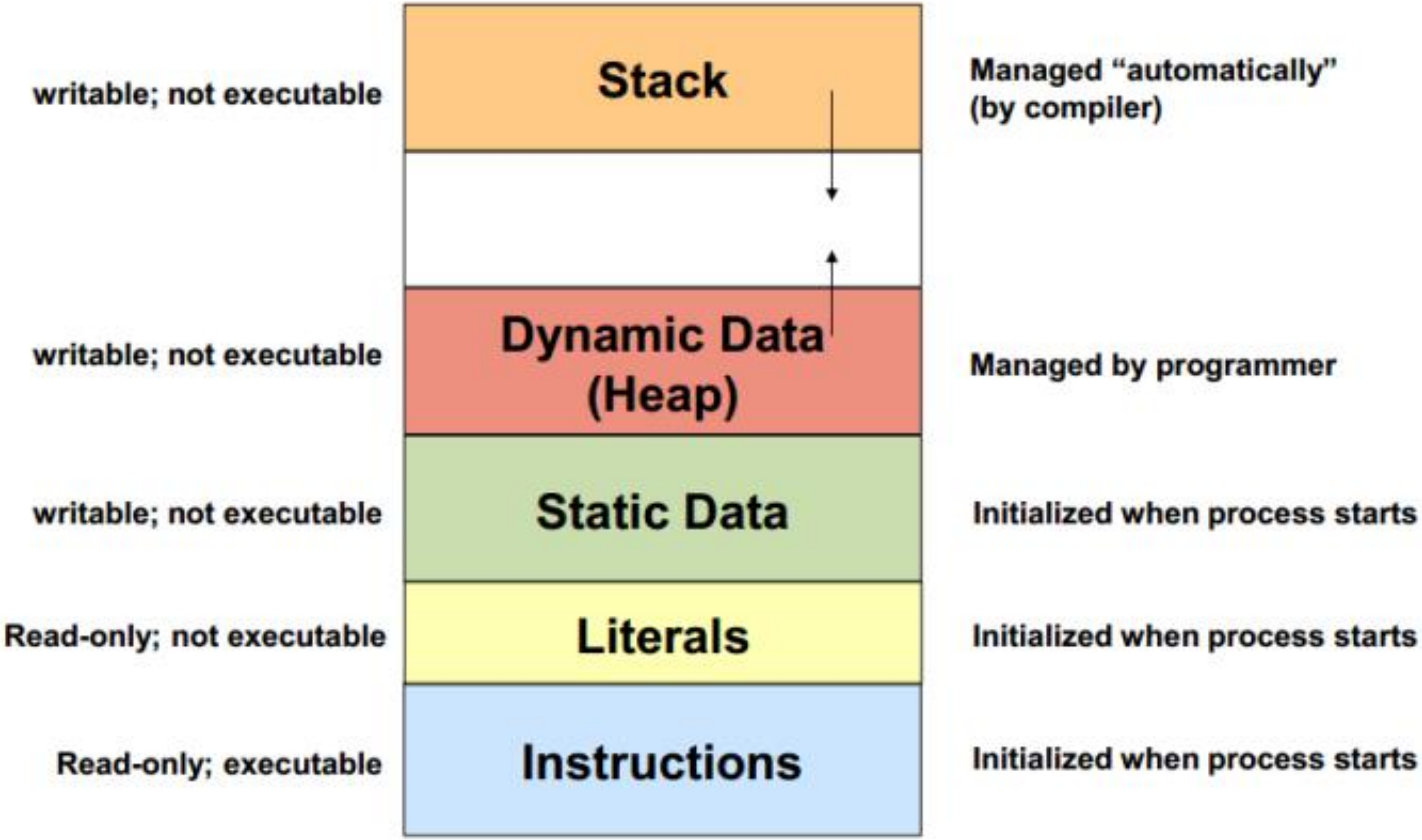
- Meaning of types
- Polymorphism and Orthogonality
- Type equivalence
- Type compatibility
- Type Inference
- Generics
- Composite Types

C++ Data Types

SECTION 3


C++ Datatypes
Built-in / Primitive
Derived
arrays
pointers
references
User-Defined
structures
unions
classes
enumerations
integer
floating
void
int
char
float
double

C++ Memory Model



Memory Addressing

Mode	Example	Meaning	When used
Register	Add R4, R3	Regs[R4]←Regs[R4]+ Regs[R3]	Value is in a register
Immediate	Add R4, #3	Regs[R4] ← Regs[R4] + 3	For constants
Displacement	Add R4, 100(R1)	Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]	Access local variables
Indirect	Add R4, (R1)	Regs[R4]←Regs[R4]+ Mem[Regs[R1]]	Pointers
Indexed	Add R3, (R1+R2)	Regs[R3]←Mem[Regs [R1]+Regs[R2]]	Traverse an array
Direct	Add R1, $1001	Regs[R1] ← Regs[R1] + Mem[1001]	Static data, address constant may be large

Memory Hierarchy

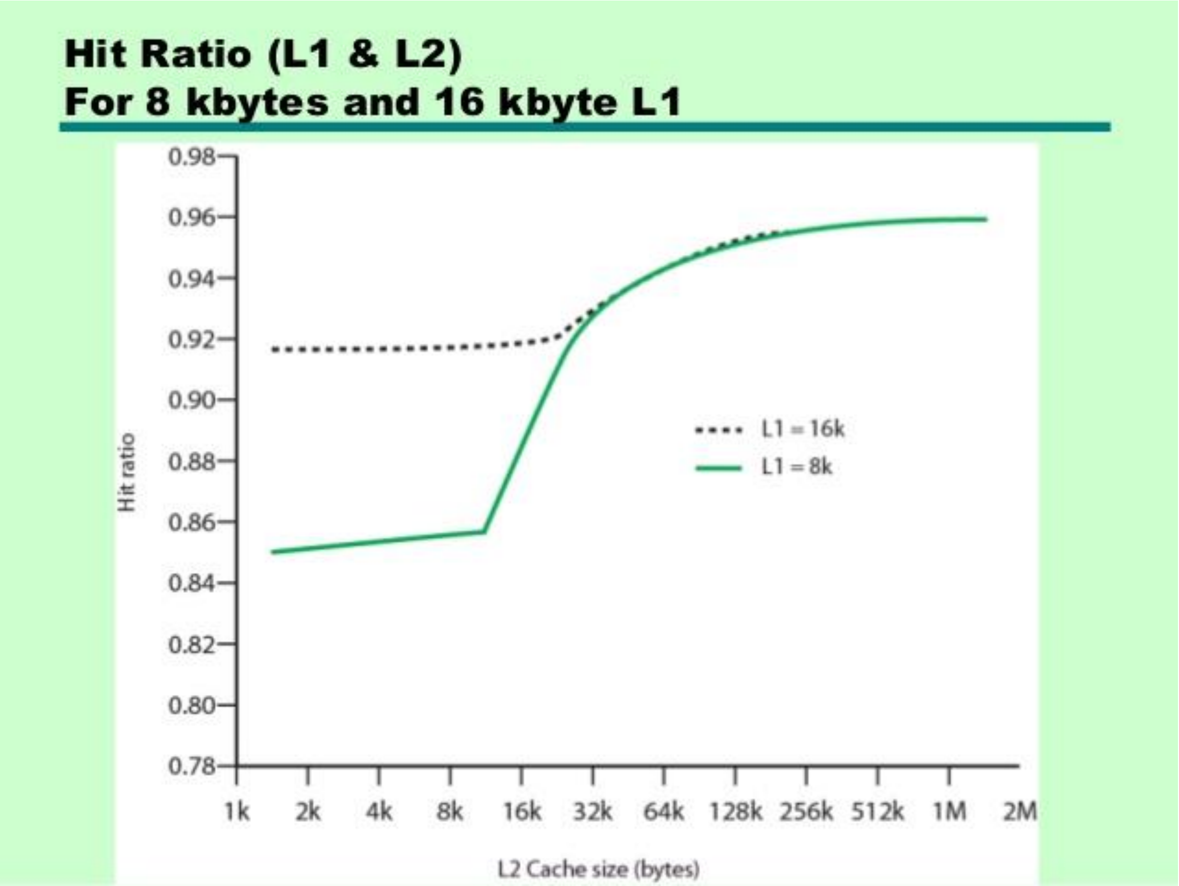
- Memory is too big to fit on one chip with a processor
  - Because memory is off-chip (in fact, on the other side of the bus), getting at it is much slower than getting at things on-chip **(Note: No longer a problem for single chip computer.)**
  - Most computers therefore employ a MEMORY HIERARCHY, in which things that are used more often are kept close at hand

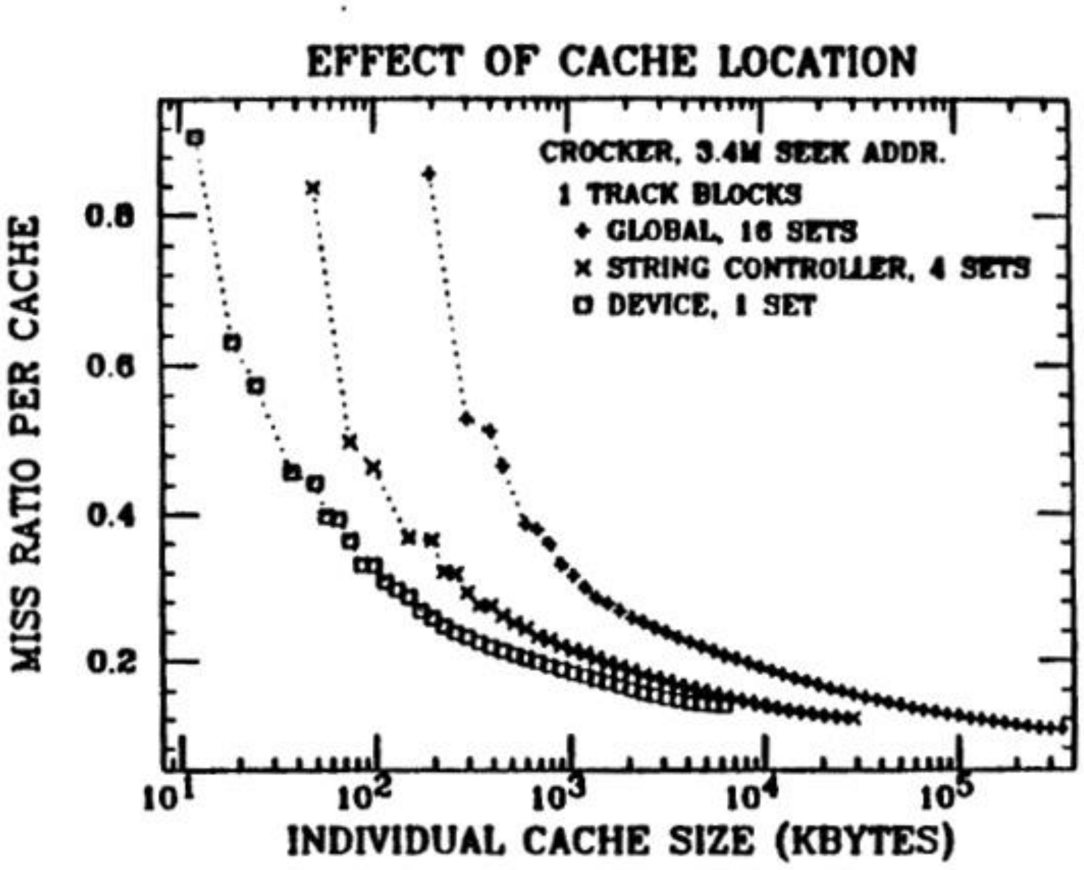
	Typical access time	Typical capacity
Registers	0.2–0.5 ns	256–1024 bytes
Primary (L1) cache	0.4–1 ns	32 K–256 K bytes
L2 or L3 (on-chip) cache	4–30 ns	1–32 M bytes
off-chip cache	10–50 ns	up to 128 M bytes
Main memory	50–200 ns	256 M–16 G bytes
Flash	40–400 μs	4 G bytes to 1 T bytes
Disk	5–15 ms	500 G bytes and up
Tape	1–50 s	effectively unlimited

Figure 5.1 **The memory hierarchy of a workstation-class computer.** Access times and capacities are approximate, based on 2015 technology. Registers are accessed within a single clock cycle. Primary cache typically responds in 1 to 2 cycles; off-chip cache in more like 20 cycles. Main memory on a supercomputer can be as fast as off-chip cache; on a workstation it is typically much slower. Flash times vary with manufacturing technology, and are longer for writes than reads. Disk and tape times are constrained by the movement of physical parts.

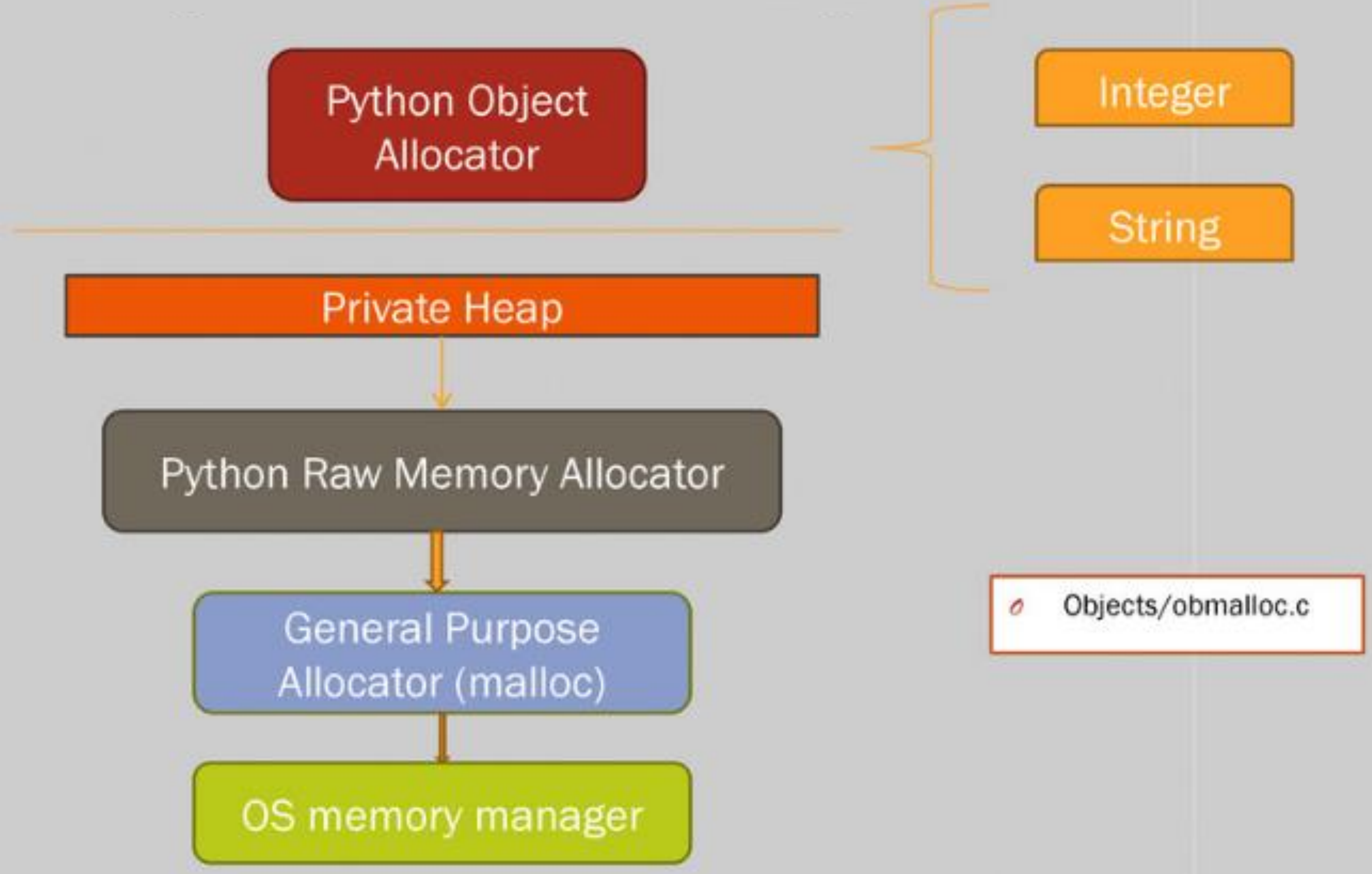
Hit Ratio and Miss Ratio of Cache

It is directly related to the performance of the processor





Python Memory Model



Python Data Types

SECTION 4

PRIMITIVE DATA TYPES IN PYTHON

STRING

SEQUENCE OF WORDS

INTEGER

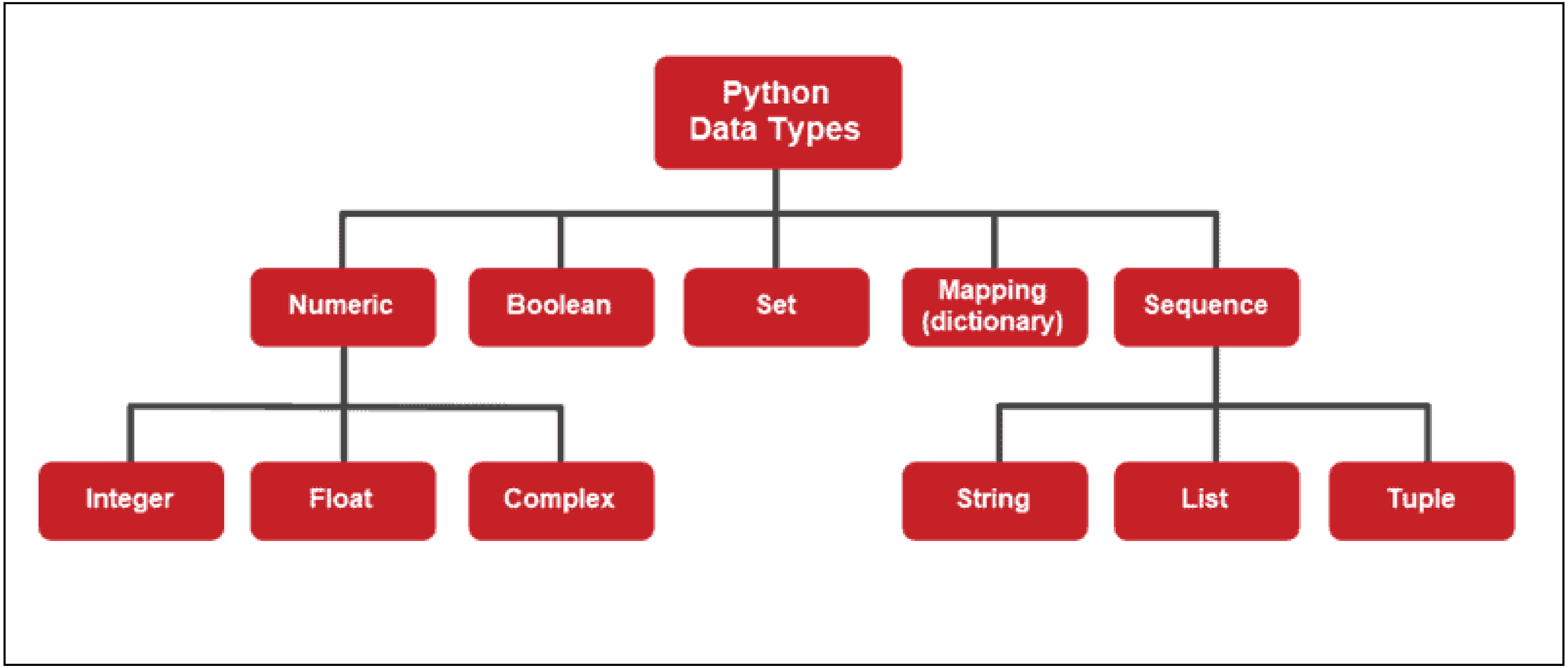
WHOLE NUMBERS

FLOAT

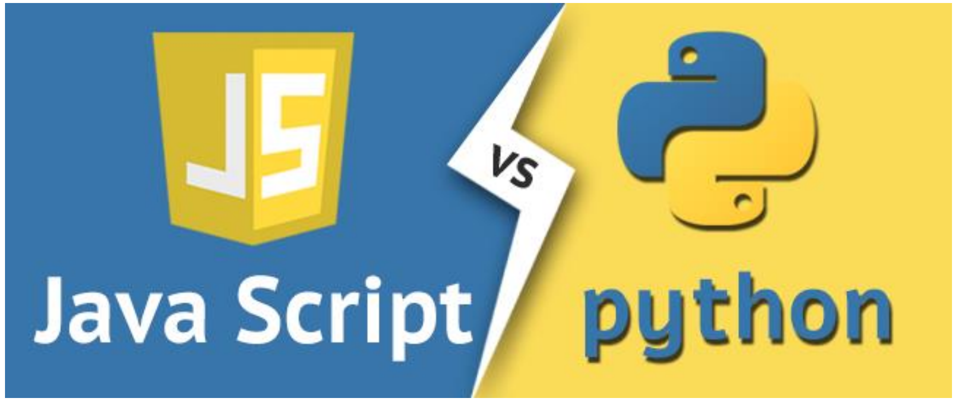
DECIMAL POINT NUMBERS

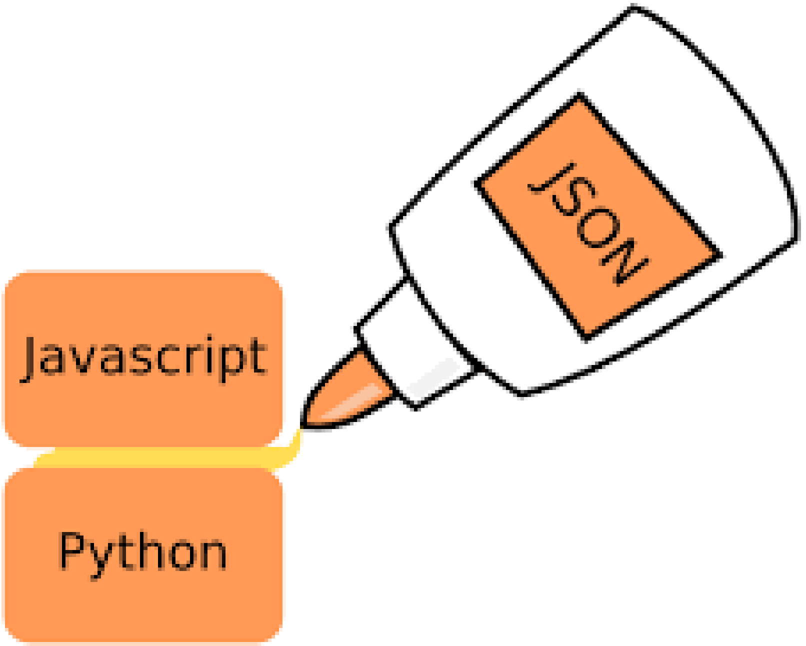
BOOLEAN

TRUE/FALSE

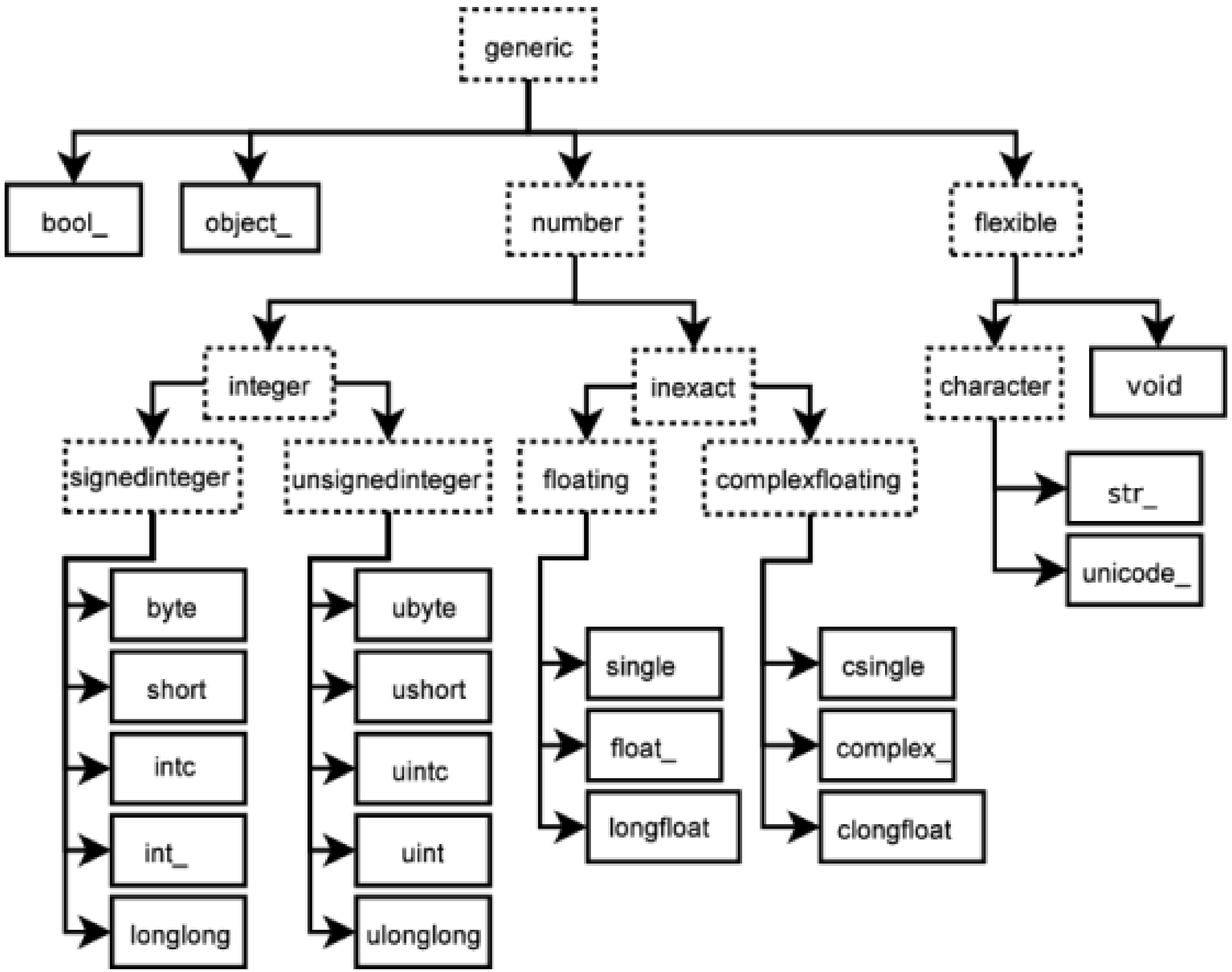


Python Data Types
Numeric
Boolean
Set
Mapping (dictionary)
Sequence
Integer
Float
Complex
String
List
Tuple





JSON Data	Python Data
String	string
Number, integer	int
Number, real	float
Boolean	bool
Array	list
Object	dict
Null	NoneType



Numpy Data Types

Expressions & Values

SECTION 5

What do programs do?

- Programs work by manipulating values
- Expressions in programs evaluate to values
  - Expression: `'a' + 'hoy'`
  - Value: `'ahoy'`
- The Python interpreter evaluates expressions and displays their values

Values

Programs manipulate values.

Each value has a certain data type.

Data type	Example values
Integers	`2 44 -3`
Floats	`3.14 4.5 -2.0`
Booleans	`True False`
Strings	`'¡hola!' 'its python time!'`

Try in a Python interpreter, like on code.cs61a.org.

Expressions (with operators)

An expression describes a computation and evaluates to a value.

Some expressions use operators:

```
18 + 69
6/23
2 * 100
2 ** 100
```

Call expressions

Many expressions use function calls:

```
pow(2, 100)
max(50, 300)
min(-1, -300)
```

Expressions (both ways)

- Expressions with operators can also be expressed with function call notation:

```
2 ** 100
pow(2, 100)
```

```
from operator import add
18 + 69
add(18, 69)
```

- The `pow()` function is a built-in; it's provided in every Python environment. Other functions (`add()` , `div()` , etc) must be imported from the `operator` module in the Python standard library.

Anatomy of a Call Expression

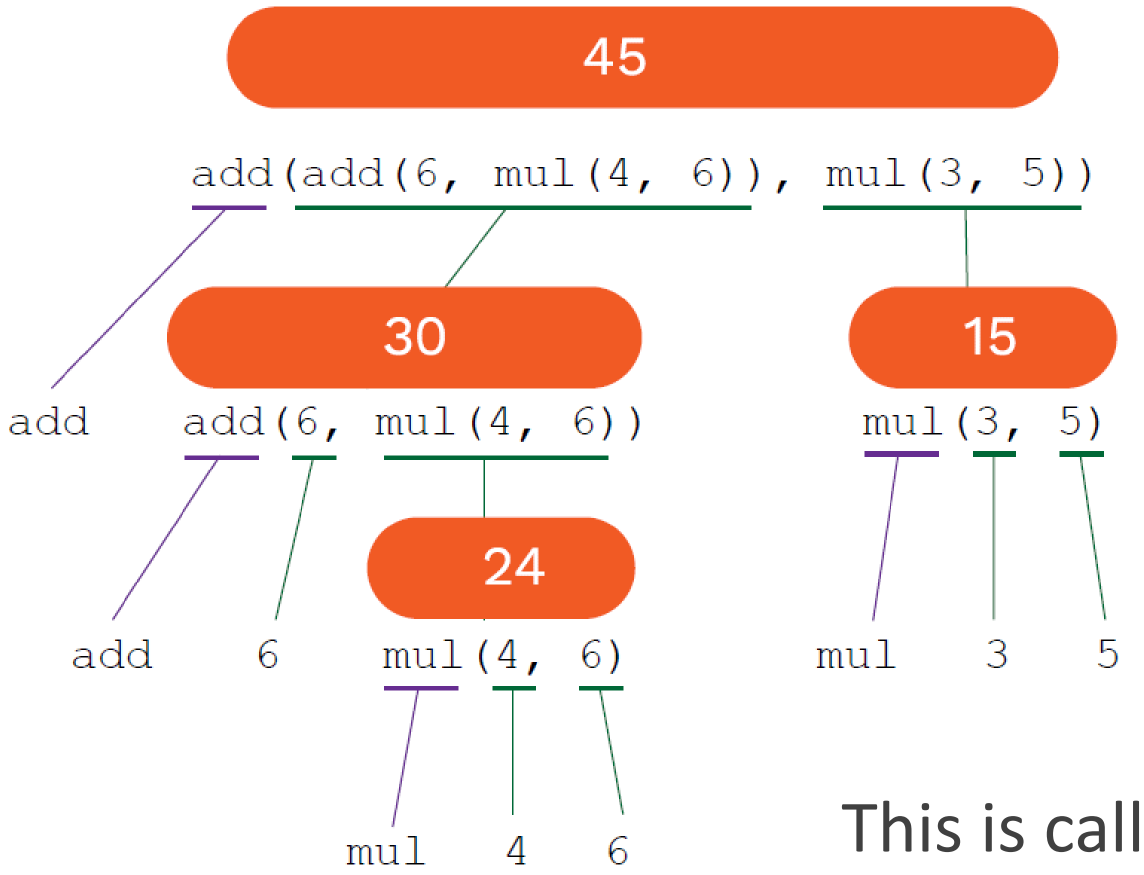
`add`	`( 18`	`,`	`69 )`
Operator	Operand		Operand

How Python evaluates a call expression:

1. Evaluate the operator
2. Evaluate the operands
3. Apply the operator (a function) to the evaluated operands (arguments)

Operators and operands are also expressions, so they must be evaluated to discover their values.

Evaluating nested expressions



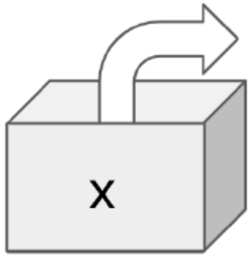
This is called an expression tree.

Names

SECTION 6

Names

A name can be bound to a value.



One way to bind a name is with an assignment statement:

```
x = 7
```

Name Value

The value can be any expression:

```
x = 1 + 2 * 3 - 4 // 5
```

Name Expression

Using names

A name can be referenced multiple times:

```
x = 10
y = 3
result1 = x * y
result2 = x + y
```

A name that's bound to a data value is also known as a variable.

Name rebinding

A name can only be bound to a single value.

```
my_name = 'Pamela'
my_name = my_name + 'ela'
```

Will that code error? If not, what will `my_name` store?

It will not error (similar code in other languages might, however). The name `my_name` is now bound to the value 'Pamelaela'.

Composite Data Types

SECTION 7

Type Systems

Composite Types

- Records (struct)
- Variant records (union)
- Arrays
  - Strings
  - Tuples
  - Arrays
  - Lists
- Sets
- Dictionary (map)
- Pointers – **l-value** [Should be Reference Type]
- Files

Python Composite Data Types

- String: “This is a string.”
- Tuple: (1, 2, 3)
- List: [1, 2, 3]
- Set: {1, 2, 3}
- Dictionary: { “A”: 1, “B”: 2, “C”: 3}

Strings

SECTION 7.1

String Topics

- String Creation (f””, r””, u””, “”)
- String Indexing
- String Slicing
- String Operations
- String Methods

The String Data Type

- The most common use of personal computers is word processing.
- Text is represented in programs by the string data type.
- A string is a sequence of characters enclosed within quotation marks (") or apostrophes (').

String and Character

Function	Description
chr()	Converts an integer to a character
ord()	Converts a character to an integer
len()	Returns the length of a string
str()	Returns a string representation of an object

```python
# string1.py

alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

print(type(alphabet))
print(len(alphabet))                 # Java's alphabet.length()
print(alphabet[9])                   # Java alphabet.charAt(9)
print(alphabet.lower())              # alphabet.toLowerCase()
print(alphabet.index("GHI"))         # alphabet.indexOf("GHI")

ch = alphabet[5]
asc = ord(ch)
print(ord(ch))                       # ascii code of the character
print(chr(asc))                      # char of the ascii code

yesLetter = ch.isalpha()              # check if ch is a letter or not
print(yesLetter)                      # Character.isLetter(ch)
yesDigit = ch.isdigit()               # Character.isDigit(ch)
print(yesDigit)
yesLetterOrNumber = ch.isalnum()      # Character.isLetterOrDigit(ch)
print(yesLetterOrNumber)
```

The String Data Type

```
>>> str1="Hello"
>>> str2='spam'
>>> print(str1, str2)
Hello spam
>>> type(str1)
<class 'str'>
>>> type(str2)
<class 'str'>
```

The String Data Type

- Getting a string as input

```
>>> firstName = input("Please enter your name: ")
Please enter your name: John
>>> print("Hello", firstName)
Hello John
```

- Notice that the input is not evaluated. We want to store the typed characters, not to evaluate them as a Python expression.

The String Data Type

- We can access the individual characters in a string through indexing.
- The positions in a string are numbered from the left, starting with 0.
- The general form is <string>[<expr>], where the value of expr determines which character is selected from the string.

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet = "Hello Bob"
>>> greet[0]
'H'
>>> print(greet[0], greet[2], greet[4])
H l o
>>> x = 8
>>> print(greet[x - 2])
B
```

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

- In a string of n characters, the last character is at position n-1 since we start counting with 0.
- We can index from the right side using negative indexes.

```
>>> greet[-1]
'b'
>>> greet[-3]
'B'
```

The String Data Type

- Indexing returns a string containing a single character from a larger string.
- We can also access a contiguous sequence of characters, called a **substring**, through a process called **slicing**.

The String Data Type

Slicing:

- `<string>[<start>:<end>]`
- `start` and `end` should both be ints
- The slice contains the substring beginning at position `start` and runs up to but **doesn't include** the position `end`.

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet[0:3]
'Hel'
>>> greet[5:9]
' Bob'
>>> greet[:5]
'Hello'
>>> greet[5:]
' Bob'
>>> greet[:]
'Hello Bob'
```

```python
# string2.py
# slicing
#           01234567890123456789012345
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# - (nega) -65432109876543210987654321

# single character access you can use array indexing way to access it.
print(alphabet[10])
#print(alphabet[27])   # it will cause index Out of bound problem.
print(alphabet[-1])
# substring
# str.substring(1, 3)
# string[start:end:step]  start:start index, end: the end index (not included), step +=
alphasub = alphabet[1:3]
print(alphasub)
alphaeven = alphabet[0:26:2]
print(alphaeven)
alphaodd = alphabet[1:26:2]
print(alphaodd)
alphaD3 = alphabet[3:26:3]
print(alphaD3)
alpharev = alphabet[-1::-1]
print(alpharev)
alphalast5B = alphabet[-1:-6:-1]
print(alphalast5B)
alphalast5F = alphabet[-5::1]
print(alphalast5F)
```

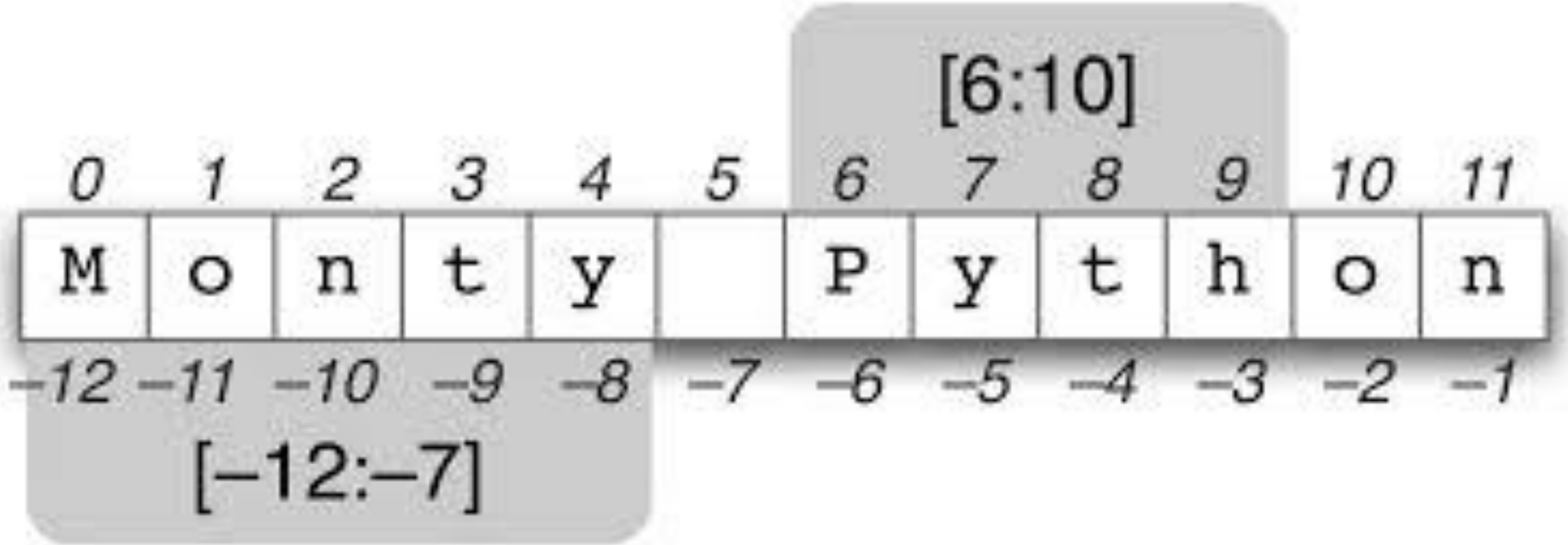
```python
# string[index] means a single character string.charAt(index)
# string[start:end] === string[start:end:1]
# string[::]        === string[0:len(string):1]
alphanothing = alphabet[::]
print(alphanothing)
# string[:2:]       === string[0:2:1] === string[0:2]
alpha02 = alphabet[:2:]
print(alpha02)
# string[-3:-1]     === string[n-3:n-1] === string[n-3:n-1:1]
alphan3n1 = alphabet[-3:-1]
print(alphan3n1)
# string[-1:-3]     === string[n-1:n-3] === string[n-1:n-3:1]
alphan1n3 = alphabet[-1:-3]
print("**"+alphan1n3+"**")
alphan1n3 = alphabet[-1:-3:-1]
print("**"+alphan1n3+"**")
# string[-1:]  === string[n-1:n] === string[n-1:n:1]
alpharev = alphabet[-1:]
print(alpharev)
alpharev2 = alphabet[-1::-2]
print(alpharev2)
```

The String Data Type

- If either expression is missing, then the start or the end of the string are used.
- Can we put two strings together into a longer string?
- Concatenation “glues” two strings together (+)
- Repetition builds up a string by multiple concatenations of a string with itself (*)

The String Data Type

- Text in a program is represented by the string data type.
- String can also be viewed as an linear (1-D) array of characters.
- String can also be viewed as a sequence of characters.



The String Data Type

- The function len will return the length of a string.

```
>>> "spam" + "eggs"
'spameggs'
>>> "Spam" + "And" + "Eggs"
'SpamAndEggs'
>>> 3 * "spam"
'spamspamspam'
>>> "spam" * 5
'spamspamspamspamspam'
>>> (3 * "spam") + ("eggs" * 5)
'spamspamspameggseggseggseggseggs'
```

The String Data Type

```
>>> len("spam")
4
>>> for ch in "Spam!":
            print (ch, end=" ")

S p a m !
```

The String Data Type

Operator	Meaning
+	Concatenation
*	Repetition
<string>[]	Indexing
<string>[:]	Slicing
len(<string>)	Length
for <var> in <string>	Iteration through characters

String Methods

Function	Meaning
s.capitalize()	Copy of s with only the first character capitalized
s.center(width)	Copy of s centered in a field of given width
s.count(sub)	Count the number of occurrences of sub in s
s.find(sub)	Find the first position where sub occurs in s
s.join(list)	Concatenate list into a string, using s as separator
s.ljust(width)	Like center, but s is left-justified
s.lower()	Copy of s in all lowercase characters

String Methods

Function	Meaning
s.lstrip()	Copy of s with leading white space removed
s.replace(oldsub, newsub)	Replace all occurrences of oldsub in s with newsub
s.rfind(sub)	Like find, but returns the rightmost position
s.rjust(width)	Like center, but s is right-justified
s.rstrip()	Copy of s with trailing white space removed
s.split()	Split s into a list of substrings (see text)
s.title()	Copy of s with first character of each word capitalized
s.upper()	Copy of s with all characters converted to uppercase.

More String Methods

- There are a number of other string methods. Try them all!
  - `s.capitalize()` – Copy of s with only the first character capitalized
  - `s.title()` – Copy of s; first character of each word capitalized
  - `s.center(width)` – Center s in a field of given width

More String Methods

- `s.count(sub)` – Count the number of occurrences of sub in s
- `s.find(sub)` – Find the first position where sub occurs in s
- `s.join(list)` – Concatenate list of strings into one large string using s as separator.
- `s.ljust(width)` – Like center, but s is left-justified

More String Methods

- `s.lower()` – Copy of s in all lowercase letters
- `s.lstrip()` – Copy of s with leading whitespace removed
- `s.replace(oldsub, newsub)` – Replace occurrences of oldsub in s with newsub
- `s.rfind(sub)` – Like find, but returns the right-most position
- `s.rjust(width)` – Like center, but s is right-justified

More String Methods

- `s.rstrip()` – Copy of s with trailing whitespace removed
- `s.split()` – Split s into a list of substrings
- `s.upper()` – Copy of s; all characters converted to uppercase

String Operations

Python

Created by Hanzel Godinez

@GodinezHanzel

General operations

split	Returns a list of substrings in the string
isalpha	Returns True or False if a string containing just alphabetic characters
isdigit	Returns True or False if a string containing just numbers
*	Replicates a string

Capitalization operations

upper	Converts a string to uppercase
lower	Converts a string to lowercase
title	Capitalizes the first letter of each word in string
capitalize	Capitalize converts only the first character of a string to uppercase

Concatenation operations

join	Takes a list of strings and puts them together to form a single string
+	Add two strings together, less efficient than join method

Searching operations

find	Searches for the target in a string, returns -1 if substring does not exists in the string
index	Searches for the target in a string, raises a ValueError exception if substring does not exists in the string
rfind	Searches for the target in a string, from right to left or from last position to 0 position
rindex	Searches for the target in a string, from right to left or from last position to 0 position
startswith	Checks the beginning of a string for a match
endswith	Checks the end of a string for a match
replace	Replaces the target with a new string
strip	Returns a new string without any whitespace or other characters at the beginning or end of the string.
rstrip	Returns a new string without any whitespace or other characters at the end of the string

```python
# string3.py

a = "abcde"
b = "abcde"
c = "abccd"
e = "bbcde"
f = "ABCDE"
g = "abcdefgh"

alphabet = "".join([chr(x) for x in range(65, 91)])

def indexOf(string, pattern, start=0):
    idx = 0
    try:
        idx = string.index(pattern, start)
    except:
        idx = -1
    return idx
```

```python
# alphanumeric by ASCII for comparison
print(a==b)   # System.out.println(a.equals(b));
print(a==c)
print(a!=c)
print(a>c)    # True:  System.out.println(a.comparTo(c)>0);
print(a>=c)   # True:  System.out.println(a.comparTo(c)>=0);
print(a<c)    # False: System.out.println(a.comparTo(c)<0);
print(a>f)    # True:  lowercase is greater than uppercase in ASCII code
print(a>g)    # Fasle: because g is longer
print(alphabet)

alpha2 = "-".join([chr(x) for x in range(97, 123)])
print(alpha2)
print("GHI" in alphabet)  # alphabet.contains("GHI")
print("ZZZ" in alphabet)
print(alphabet.index("GHI"))
source = "AABDBACDEDABAAABBABABAABBBBBAAAA"
ablist = []   # creating a list: array, arraylist
pos = indexOf(source, "AB")
while pos>0:
    ablist.append(pos)
    pos = indexOf(source, "AB", pos+2)
print(ablist)
```

Format Codes (Builtins)

- For builtins, there are standard format codes

Old Format	New Format	Description
"%d"	"d"	Decimal Integer
"%f"	"f"	Floating point
"%s"	"s"	String
"%e"	"e"	Scientific notation
"%x"	"x"	Hexadecimal

- Plus there are some brand new codes

	"o"	Octal
	"b"	Binary
	"%"	Percent

Formatted String

The %3d specifier means a minimum width of three spaces, which, by default, will be right-justified:

"%3d“ % 0	0
"%3d“ % 123456789	123456789
"%3d“ % -10	-10
"%3d“ % -123456789	-123456789

Left-justifying Formatted String

To left-justify integer output with printf, just add a minus sign (-) after the % symbol, like this:

"%-3d" % 0	0
"%-3d" % 123456789	123456789
"%-3d" % -10	-10
"%-3d" % -123456789	-123456789

The Formatted String zero-fill option

To zero-fill your printf integer output, just add a zero (0) after the % symbol, like this:

"%03d" % 0	000
"%03d" % 1	001
"%03d" % 123456789	123456789
"%03d" % -10	-10
"%03d" % -123456789	-123456789

Integer Formatted String

As a summary of printf integer formatting, here’s a little collection of integer formatting examples. Several different options are shown, including a minimum width specification, left-justified, zero-filled, and also a plus sign for positive numbers.

Description	Code	Result
At least five wide	"'%5d'" % 10	' 10'
At least five-wide, left-justified	"'%-5d'" % 10	'10 '
At least five-wide, zero-filled	"'%05d'" % 10	'00010'
At least five-wide, with a plus sign	"'%+5d'" % 10	' +10'
Five-wide, plus sign, left-justified	"'%-+5d'" % 10	'+10 '

Formatted String - floating point numbers

Here are several examples showing how to format floating-point numbers with printf:

Description	Code	Result
Print one position after the decimal	"'%.1f'" % 10.3456	'10.3'
Two positions after the decimal	"'%.2f'" % 10.3456	'10.35'
Eight-wide, two positions after the decimal	"'%8.2f'" % 10.3456	' 10.35'
Eight-wide, four positions after the decimal	"'%8.4f'" % 10.3456	' 10.3456'
Eight-wide, two positions after the decimal, zero-filled	"'%08.2f'" % 10.3456	'00010.35'
Eight-wide, two positions after the decimal, left-justified	"'%-8.2f'" % 10.3456	'10.35 '
Printing a much larger number with that same format	"'%-8.2f'" % 101234567.3456	'101234567.35'

Formatted String

Here are several examples that show how to format string output with printf:

Description	Code	Result
A simple string	"'%s'" % "Hello"	`'Hello'`
A string with a minimum length	"'%10s'" % "Hello"	`'     Hello'`
Minimum length, left-justified	"'%-10s'" % "Hello"	`'Hello     '`

```python
# string4.py
#  formatted strings
points = [(1, 2), (3,4), (-1, 2), (5, 1), (1, -4)]

for p in points:
    s = "Point(%d, %d)" % p
    print(s)

name = "Eric"
age = 15
score = 10
s1 = (name, age, score)
print("Student: %s, Age:%d, Score:%d" % (name, age, score))
print("Student: %s, Age:%d, Score:%d" % s1)

f = 98
c = (f-32)/1.8
print("%.2fF is %.2fC" % (f, c))
```

Tuples

SECTION 7.2

Tuple

- A tuple is a list that cannot change. Python refers to a value that cannot change as **immutable**. So by definition, **a tuple is an immutable list.**

Basic Tuples Operations

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Tuple

- Defining a tuple
- Defining a tuple that has one element
- Assigning a tuple

```python
# tuple1.py
#  fixed size sequence - closer to Java array
#  tuples are immutable (no change)
#  fixed data vector (no change)

def swap(a, b): return (b, a)

def gcd(m, n):
    if (n==0): return m
    return gcd(n, m%n)

def gcf(m, n):
    while n!=0: (m, n) = (n, m%n)
    return m

t = (1, 2, 3)
for i in range(len(t)):
    print(t[i])

#t.append(4)  # no, no no no. tuple is of fixes size
print(t[2])
```

```
tx = (t[0], t[1], 7)
print(tx)

a = 3
b = 4
print("a=", a, "b=", b)
(a, b) = swap(a, b)
print("a=", a, "b=", b)

print(gcf(32, 48))
print(gcf(65, 78))
```

Lists

SECTION 7.3

List and String

- String to List
- List to String

```python
# list1.py
#  lists are of flexible length. They are equivalent to Java ArrayLists
a = ['a', 'c', 'e']
print(a)

s = "".join(a)
print(s)

a2 = list(s)
print(a2)

b = ["I", 'Love', "You"]
punc =". "
question = "?"
s2 = " ".join(b)
s3 = s2+punc
print(s3)
s4 = s2+question
print(s4)

b2 = s2.split(" ")
print(b2)
```

Lists and Arrays

- Python lists are ordered sequences of items. For instance, a sequence of *n* numbers might be called *S*:
  $S = s_0, s_1, s_2, s_3, \ldots, s_{n-1}$
- Specific values in the sequence can be referenced using *subscripts*.
- By using numbers as subscripts, mathematicians can succinctly summarize computations over items in a sequence using subscript variables.

$$\sum_{i=0}^{n-1} s_i$$

Lists and Arrays

- Suppose the sequence is stored in a variable `s`. We could write a loop to calculate the sum of the items in the sequence like this:

```
sum = 0
for i in range(n):
    sum = sum + s[i]
```

- Almost all computer languages have a sequence structure like this, sometimes called an *array*.

Lists and Arrays

- A list or array is a sequence of items where the entire sequence is referred to by a single name (i.e. `s`) and individual items can be selected by indexing (i.e. `s[i]`).
- In other programming languages, arrays are generally a fixed size, meaning that when you create the array, you have to specify how many items it can hold.
- Arrays are generally also *homogeneous*, meaning they can hold only one data type.

Lists and Arrays

- Python lists are dynamic. They can grow and shrink on demand.
- Python lists are also *heterogeneous*, a single list can hold arbitrary data types.
- Python lists are mutable sequences of arbitrary objects.

```
#    vector in C++, arraylist in Java, same as array? no no no
#    list is dynamic, flexible
from random import *
import numpy as np

# exmaple list
a = [1, 2, 3]
print(a)

# Comprehesion list
b = [x**2 for x in range(1, 4)]
print(b)

# loop for list create
c = []
for i in range(len(a)):
    z = a[i]-2
    c.append(z)

print(c)
```

```python
d = [0 for i in range(10)]
for i in range(10):
    d[i] = randint(1, 10)    # from 1, to 10 (included)
print(d)

e = list(np.zeros(10))    # np.zeros(10) create 10 element 0 array
print(e)
```

List Operations

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration
<expr> in <seq>	Membership (Boolean)

List Operations

- Except for the membership check, we've used these operations before on strings.
- The membership operation can be used to see if a certain value appears anywhere in a sequence.

```
>>> lst = [1,2,3,4]
>>> 3 in lst
True
```

List Operations

- The summing example from earlier can be written like this:

```
sum = 0
for x in s:
    sum = sum + x
```

- Unlike strings, lists are mutable:

```
>>> lst = [1,2,3,4]
>>> lst[3]
4
>>> lst[3] = "Hello"
>>> lst
[1, 2, 3, 'Hello']
>>> lst[2] = 7
>>> lst
[1, 2, 7, 'Hello']
```

List Operations

- A list of identical items can be created using the repetition operator. This command produces a list containing 50 zeroes:

```
zeroes = [0] * 50
```

List Operations

- Lists are often built up one piece at a time using append.

```
nums = []
x = float(input('Enter a number: '))
while x >= 0:
    nums.append(x)
    x = float(input('Enter a number: '))
```

- Here, `nums` is being used as an accumulator, starting out empty, and each time through the loop a new value is tacked on.

List Operations

Method	Meaning
<list>.append(x)	Add element x to end of list.
<list>.sort()	Sort (order) the list. A comparison function may be passed as a parameter.
<list>.reverse()	Reverse the list.
<list>.index(x)	Returns index of first occurrence of x.
<list>.insert(i, x)	Insert x into list at index i.
<list>.count(x)	Returns the number of occurrences of x in list.
<list>.remove(x)	Deletes the first occurrence of x in list.
<list>.pop(i)	Deletes the i^{th} element of the list and returns its value.

List Operations

```
>>> lst = [3, 1, 4, 1, 5, 9]
>>> lst.append(2)
>>> lst
[3, 1, 4, 1, 5, 9, 2]
>>> lst.sort()
>>> lst
[1, 1, 2, 3, 4, 5, 9]
>>> lst.reverse()
>>> lst
[9, 5, 4, 3, 2, 1, 1]
>>> lst.index(4)
2
```

```
>>> lst.insert(4, "Hello")
>>> lst
[9, 5, 4, 3, 'Hello', 2, 1, 1]
>>> lst.count(1)s
2
>>> lst.remove(1)
>>> lst
[9, 5, 4, 3, 'Hello', 2, 1]
>>> lst.pop(3)
3
>>> lst
[9, 5, 4, 'Hello', 2, 1]
```

List Operations

- Most of these methods don't return a value – they change the contents of the list in some way.
- Lists can grow by appending new items, and shrink when items are deleted. Individual items or entire slices can be removed from a list using the `del` operator.

List Operations

```
>>> myList=[34, 26, 0, 10]
>>> del myList[1]
>>> myList
[34, 0, 10]
>>> del myList[1:3]
>>> myList
[34]
```

- `del` isn't a list method, but a built-in operation that can be used on list items.

List Operations

- Basic list principles
  - A list is a sequence of items stored as a single object.
  - Items in a list can be accessed by indexing, and sublists can be accessed by slicing.
  - Lists are mutable; individual items or entire slices can be replaced through assignment statements.

List Operations

- Lists support a number of convenient and frequently used methods.
- Lists will grow and shrink as needed.

```
# list3.py
from pylab import *

def indexOf(alist, key):
    idx = -1
    try: idx = alist.index(key)
    except: pass
    return idx

a = [1, 2]
b = [3, 4]

print(a+b)   # concatenation

c = [1, 2] * 3
print(c)     # duplication

d = [1, 2, 3, 4, 5, 6, 1, 2]
b1 = 6 in d
print("6 in d is ", b1)
b2 = 15 in d
print("15 in d is ", b2)
```

```
# aggregate function
s = sum(d)
print("sum(d)=", s)
s = max(d)
print("max(d)=", s)
s = min(d)
print("min(d)=", s)

# sort: destructive sorting, d got changed
d.sort()
print(d)

# sorted is non-destructive sorting, a new list sorted will be created
d = [1, 2, 3, 4, 5, 6, 1, 2]
f = sorted(d)
print("d[]=", d)
print("f[]=", f)

d = [1, 3, 5, 6, 7, 8, 10]
i = indexOf(d, 7)
print("7 is at index %3d" % i)  # similar to System.out.printf();
i = indexOf(d, 4)
print("4 is at inddex %d" % i)
```

List and Numpy Array

- **Numpy (Links to an external site.)** is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the *rank* of the array; the *shape* of an array is a tuple of integers giving the size of the array along each dimension.

List and Numpy Array

- The Python core library provided **Lists**. A list is the Python equivalent of an array, but is resizeable and can contain elements of different types.
- A common beginner question is what is the real difference here. The answer is performance. Numpy data structures perform better in:
  - **Size** - Numpy data structures take up less space
  - **Performance** - they have a need for speed and are faster than lists
  - **Functionality** - SciPy and NumPy have optimized functions such as linear algebra operations built in.

```python
# list4.py
from pylab import *
import numpy as np

a = np.linspace(0, 10, 11) # creating sample
#a = [1, 2, 4, 8, 16]
asq = [x**2 for x in a]
acube = [x**3 for x in a]
print(a)
print(asq)
figure()
bar(a, asq, color=(1.0, 0, 0, 1)) # R G B
show()

figure()
bar(a, acube, color=(0, 0.5, 0.5, 1)) # R G B
show()
```

Comprehension List

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

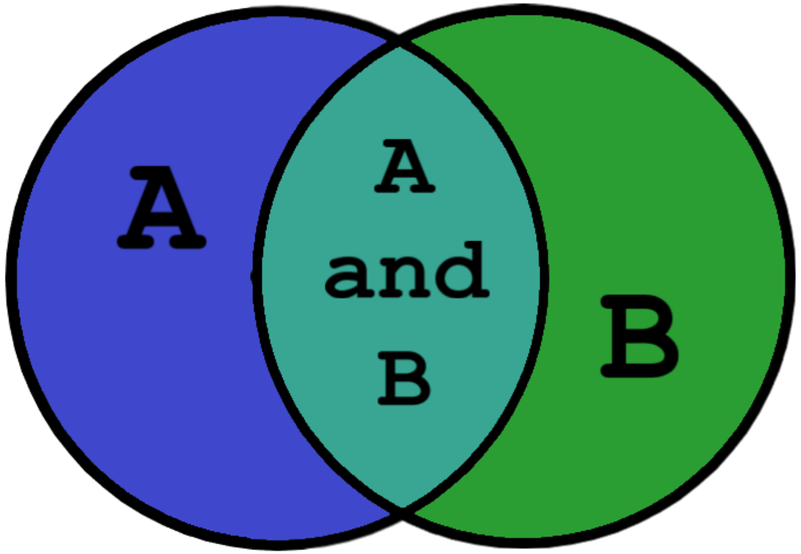
```python
# list5.py
a = ["I", "am", "a", "good", "student", "from", "Washington", "high", "school"]
first = [word[0] for word in a]
s = "".join(first)
print(s)

print("".join([word[0] for word in ["I", "am", "a", "good", "student", "from",
"Washington", "high", "school"]]))

last = [word[-1] for word in a]
s ="".join(last)
print(s)
middle = [word[len(word)//2] for word in a]
s = "".join(middle)
print(s)
```

Sets

SECTION 7.4

Set

Perhaps you recall learning about **sets** and **set theory** at some point in your mathematical education. Maybe you even remember Venn diagrams:



Set

In mathematics, a rigorous definition of a set can be abstract and difficult to grasp. Practically though, a set can be thought of simply as a well-defined collection of distinct objects, typically called elements or members.

Defining a Set

Set Size and Membership

Operating on a Set

Available Operators and Methods

Operation	Equivalent	Result
len(s)		cardinality of set s
x in s		test x for membership in s
x not in s		test x for non-membership in s
s.issubset(t)	s <= t	test whether every element in s is in t
s.issuperset(t)	s >= t	test whether every element in t is in s
s.union(t)	s | t	new set with elements from both s and t
s.intersection(t)	s & t	new set with elements common to s and t
s.difference(t)	s - t	new set with elements in s but not in t
s.symmetric_difference(t)	s ^ t	new set with elements in either s or t but not both
s.copy()		new set with a shallow copy of s

Modifying a Set

```python
# non-recurring
# { }

s = {4, 3, 1} | {3}  # empty set union with {3}  equivalent to add 3 to s
print(s)

s = {1, 2, 3, 4, 5}
t = {2, 4}

print(s | t)    # union
print(s & t)    # intersect
print(s - t)    # set difference

slist = list(s)
print(slist)
ss = set(slist)
print(ss)

a = [1, 2, 3, 3, 1, 2, 1, 3, 1, 1, 2, 4, 5, 6]
s = set(a)
print(s)
a = list(s)
print(a)
```

Dictionary

SECTION 7.5

Dictionary Basics

- Lists allow us to store and retrieve items from sequential collections.
- When we want to access an item, we look it up by index – its position in the collection.
- What if we wanted to look students up by student id number? In programming, this is called a *key-value pair*
- We access the value (the student information) associated with a particular key (student id)

Dictionary Basics

- Three are lots of examples!
  - Names and phone numbers
  - Usernames and passwords
  - State names and capitals
- A collection that allows us to look up information associated with arbitrary keys is called a *mapping*.
- Python dictionaries are *mappings*. Other languages call them *hashes* or *associative arrays*.

Dictionary Basics

- Dictionaries can be created in Python by listing key-value pairs inside of curly braces.
- Keys and values are joined by ":" and are separated with commas.

```
>>>passwd = {"guido":"superprogrammer",
"turing":"genius", "bill":"monopoly"}
```

- We use an indexing notation to do lookups

```
>>> passwd["guido"]
'superprogrammer'
```

Dictionary Basics

`<dictionary>[<key>]` returns the object with the associated key.

- Dictionaries are mutable.

```
>>> passwd["bill"] = "bluescreen"
>>> passwd
{'guido': 'superprogrammer', 'bill': 'bluescreen',
'turing': 'genius'}
```

- Did you notice the dictionary printed out in a different order than it was created?

Dictionary Basics

- Mappings are inherently unordered.
- Internally, Python stores dictionaries in a way that makes key lookup very efficient.
- When a dictionary is printed out, the order of keys will look essentially random.
- If you want to keep a collection in a certain order, you need a sequence, not a mapping!
- Keys can be any immutable type, values can be any type, including programmer-defined.

Dictionary Operations

- Like lists, Python dictionaries support a number of handy built-in operations.
- A common method for building dictionaries is to start with an empty collection and add the key-value pairs one at a time.

```
passwd = {}
for line in open('passwords', 'r'):
    user, pass = line.split()
    passwd[user] = pass
```

Dictionary Operations

Method	Meaning
<key> in <dict>	Returns true if dictionary contains the specified key, false if it doesn’t.
<dict>.keys()	Returns a sequence of keys.
<dict>.values()	Returns a sequence of values.
<dict>.items()	Returns a sequence of tuples (key, value) representing the key-value pairs.
del <dict>[<key>]	Deletes the specified entry.
<dict>.clear()	Deletes all entries.
for <var> in <dict>:	Loop over the keys.
<dict>.get(<key>, <default>)	If dictionary has key returns its value; otherwise returns default.

Dictionary Operations

```
>>> list(passwd.keys())
['guido', 'turing', 'bill']
>>> list(passwd.values())
['superprogrammer', 'genius', 'bluescreen']
>>> list(passwd.items())
[('guido', 'superprogrammer'), ('turing', 
'genius'), ('bill', 'bluescreen')]
>>> "bill" in passwd
True
>>> "fred" in passwd
False
```

Dictionary Operations

```
>>> passwd.get('bill','unknown')
'bluescreen'
>>> passwd.get('fred','unknown')
'unknown'
>>> passwd.clear()
>>> passwd
{}
```

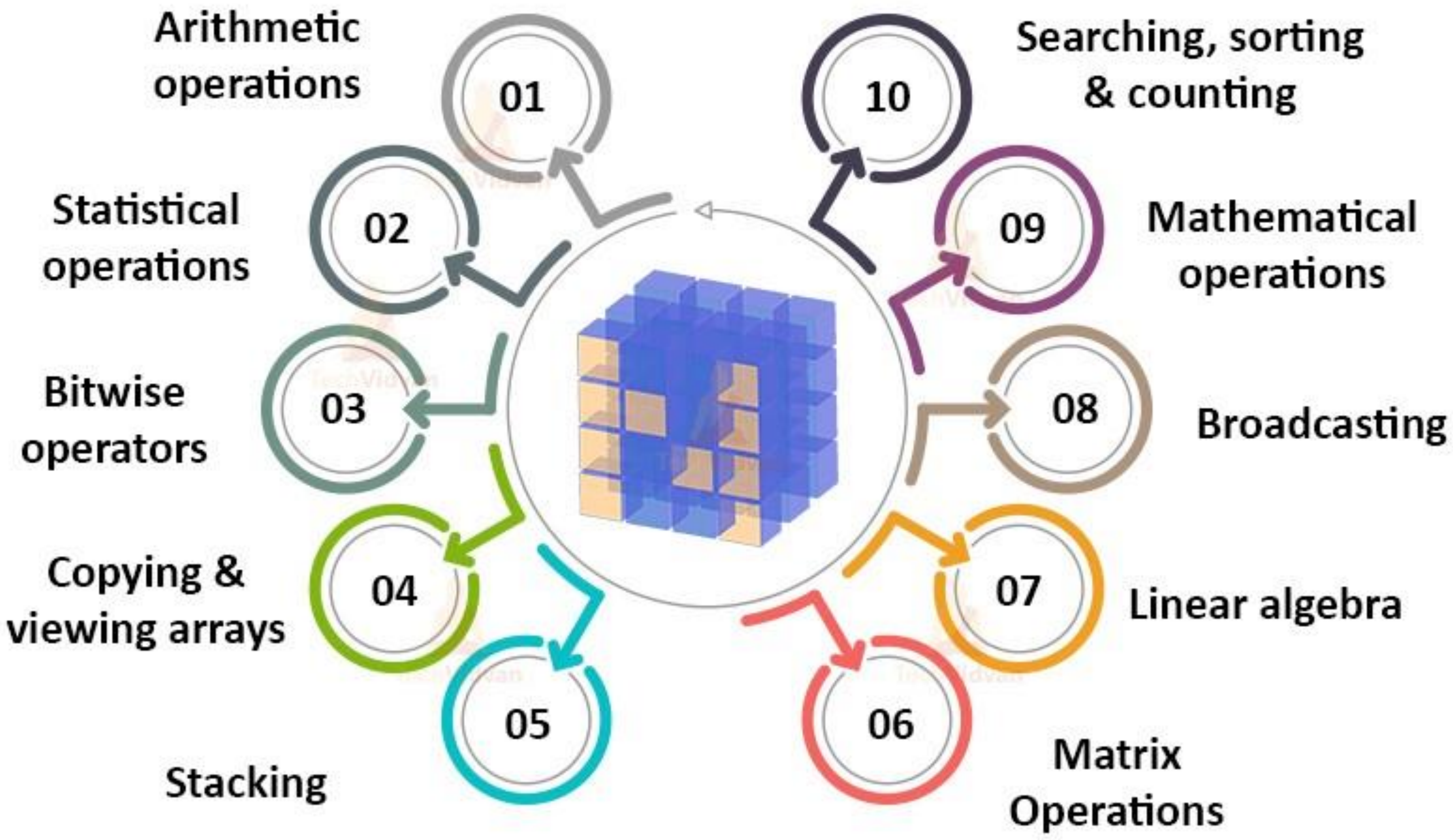
```python
from pprint import *
student = {
  'name': 'eric',
  'age': 15,
  'score': 10
}

print(student)

slist = [student, student, student]
pprint(slist)

dd = dict()
pprint(dd)
dd['key'] = "a tool to open door"
dd['cup'] = "a tool to drink water"
dd['money'] = "a tool to price things"
pprint(dd)

dd['subject'] = ['math', 'english', 'science']
dd['score'] = [10, 10, 10]
pprint(dd)
```

```python
ascii = dict()

for ch in "".join([chr(x) for x in range(65, 91)]):
    ascii[ch] = ord(ch)

print(ascii)

# ascii.items create a list of tuple of (key, value)
for item in ascii.items():
    print(item)

# ascii.keys create a list of keys
for key in ascii.keys():
    print(key)

# ascii.values create a list of values
for v in ascii.values():
    print(v)

# get each property
for k in ascii:
    print("%s - %d" % (k, ascii[k]))
```

Numpy Data Types

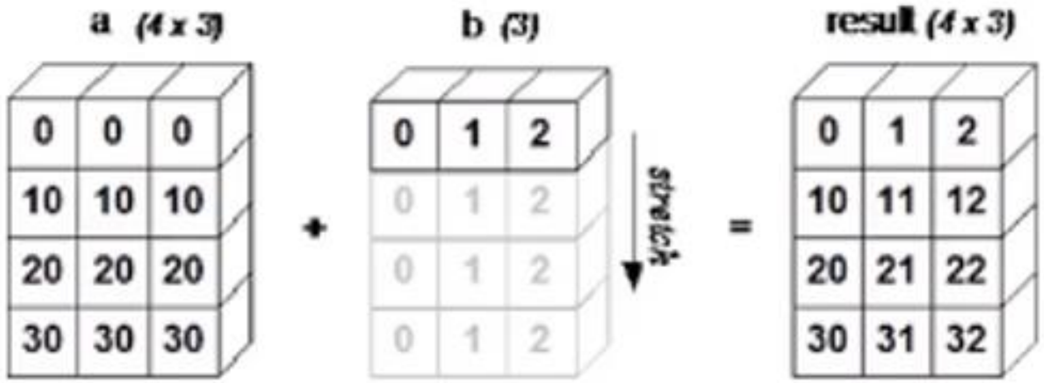
SECTION 8 (OPTIONAL)

Uses of NumPy



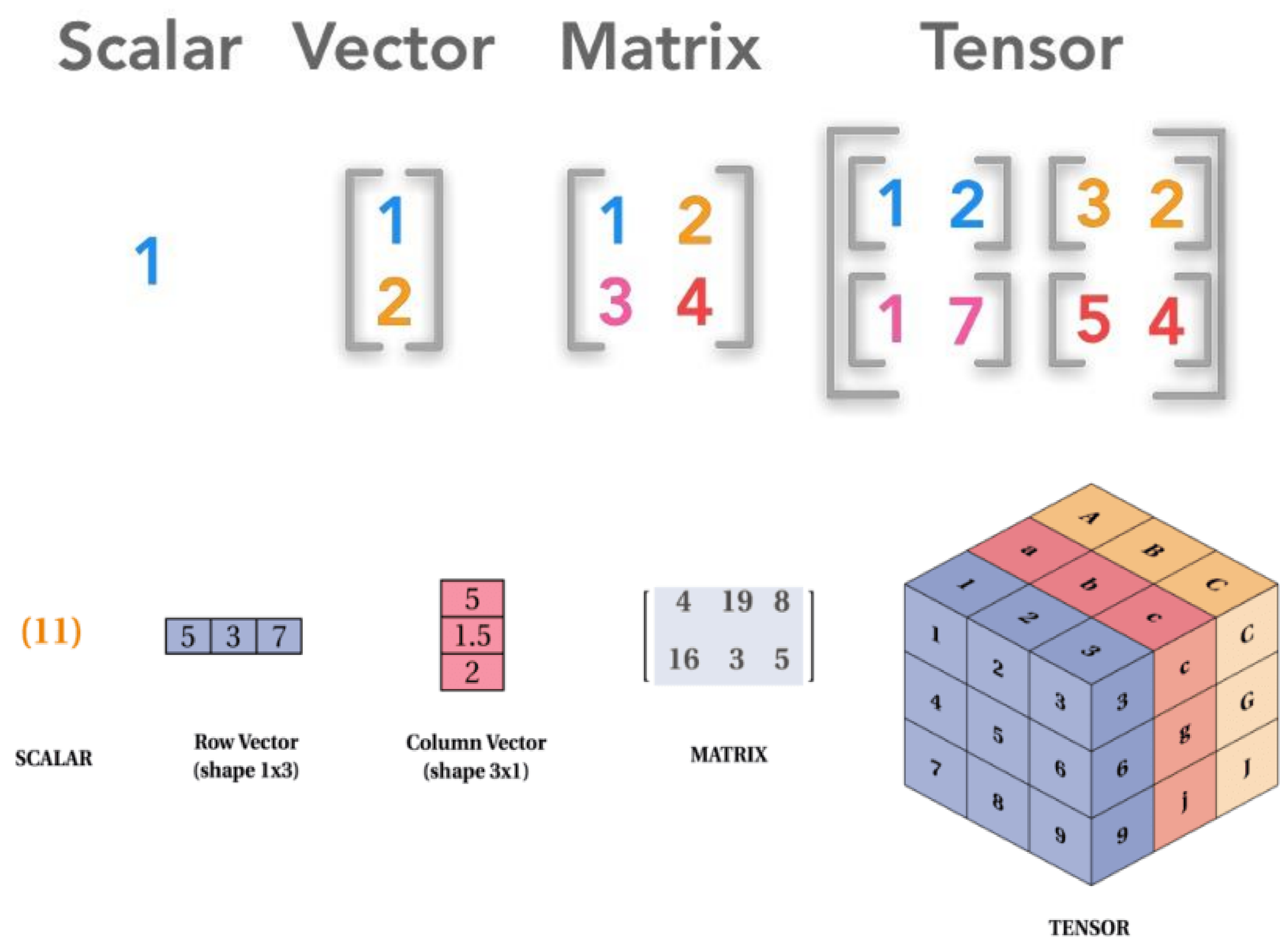
```
In [20]: import numpy as np
         a = np.array([[0,0,0],[10,10,10],[20,20,20],[30,30,30]])
         b = np.array([0,1,2])

         print('First array:\n',a,'\n')
         print('Second array:\n',b,'\n')

         print('First Array + Second Array \n',a+b)
```

```
First array:
 [[ 0  0  0]
 [10 10 10]
 [20 20 20]
 [30 30 30]]

Second array:
 [0 1 2]

First Array + Second Array
 [[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```



Array

SECTION 8.1



Scalar
Vector
Matrix
Tensor
1
1 2
1 2 3 4
1 2 3 2 1 7 5 4
(11)
5 3 7
5 1.5 2
4 19 8 16 3 5
SCALAR
Row Vector (shape 1x3)
Column Vector (shape 3x1)
MATRIX
A B C a b c 1 2 3
1 2 3 4 5 6 7 8 9
c g j C G J
TENSOR

Numpy Array

We'll cover a few categories of basic array manipulations here:

- **Attributes of arrays**: Determining the size, shape, memory consumption, and data types of arrays
- **Indexing of arrays**: Getting and setting the value of individual array elements
- **Slicing of arrays**: Getting and setting smaller subarrays within a larger array
- **Reshaping of arrays**: Changing the shape of a given array
- **Joining and splitting of arrays**: Combining multiple arrays into one, and splitting one array into many

Data Representation

```
data = np.array([1,2,3])
```

Attriubutes

- **ndim:** dimension
- **shape:** sizes for each dimension (width, length, height, plane)
- **size:** net number of elements
- **dtype:** data type
- **itemsize:** total number of items
- **nbytes:** total number of bytes

Numpy Array Attributes

```
>>> import numpy as np
>>> a = np.arange(6)        # NumPy arange returns an array object
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a = a.reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> a.shape
(2, 3)                      # note: this returns a tuple
>>> a.ndim
2
>>> a.size
6
```

Array Creation

Demo Program: array1_Creation1.py

```
import numpy as np
a = np.arange(6)
print("1-D: ", a)
a.reshape(2, 3)
print("2-D: ", a)
print("Shape: ", a.shape)
print("Dimension: ", a.ndim)
print("Size: ", a.size)
```

```
1-D:  [0 1 2 3 4 5]
2-D:  [0 1 2 3 4 5]
Shape:  (6,)
Dimension:  1
Size:  6
```

Creating Arrays

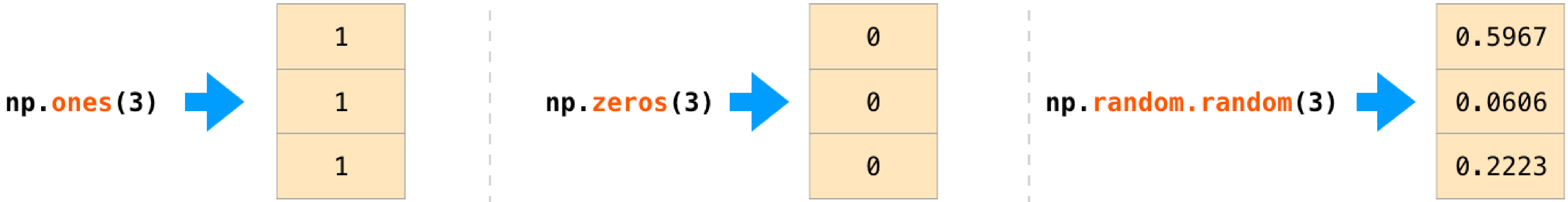
Command

```
np.array([1,2,3])
```

NumPy Array

1
2
3

Creating Arrays



Example

In [1]:
```
import numpy as np
np.random.seed(0)  # seed for reproducibility

x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

Each array has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array):

Example

In [2]:
```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

In [3]:
```
print("dtype:", x3.dtype)
```

```
dtype: int64
```

In [4]:
```
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
```

```
itemsize: 8 bytes
nbytes: 480 bytes
```

Array Creation

Demo Program: array1_Creation2.py

```
import numpy as np
np.random.seed(0)
x1 = np.random.randint(10, size=6)
x2 = np.random.randint(10, size=(3, 4))
x3 = np.random.randint(10, size=(3, 4, 5))
print("x3 ndim:", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size:", x3.size)
print("x3 dtype:", x3.dtype)
print("x3 itemsize:", x3.itemsize)
print("x3 nbytes:", x3.nbytes)
```

```
x3 ndim: 3
x3 shape: (3, 4, 5)
x3 size: 60
x3 dtype: int32
x3 itemsize: 4
x3 nbytes: 240
```

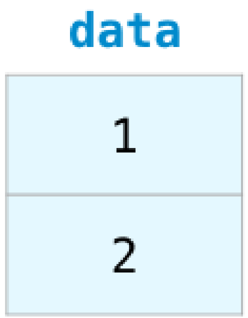
Name	Description	Syntax
empty()	Return a new array of given shape and type, without initializing entries.	empty(shape[, dtype, order])
empty_like	Return a new array with the same shape and type as a given array.	empty_like(a[, dtype, order, subok])
eye()	Return a 2-D array with ones on the diagonal and zeros elsewhere.	eye(N[, M, k, dtype])
identity()	Return the identity array.	identity(n[, dtype])
ones()	Return a new array of given shape and type, filled with ones.	ones(shape[, dtype, order])
ones_like	Return an array of ones with the same shape and type as a given array.	ones_like(a[, dtype, order, subok])
zeros	Return a new array of given shape and type, filled with zeros.	zeros(shape[, dtype, order])
zeros_like	Return an array of zeros with the same shape and type as a given array.	zeros_like(a[, dtype, order, subok])
full()	Return a new array of given shape and type, filled with fill_value.	full(shape, fill_value[, dtype, order])
full_like()	Return a full array with the same shape and type as a given array.	full_like(a, fill_value[, dtype, order, subok])

Array Arithmetic

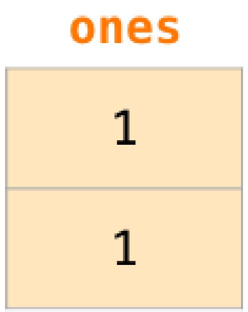
SECTION 8.2

Array Arithmetic

```
data = np.array([1,2])
```

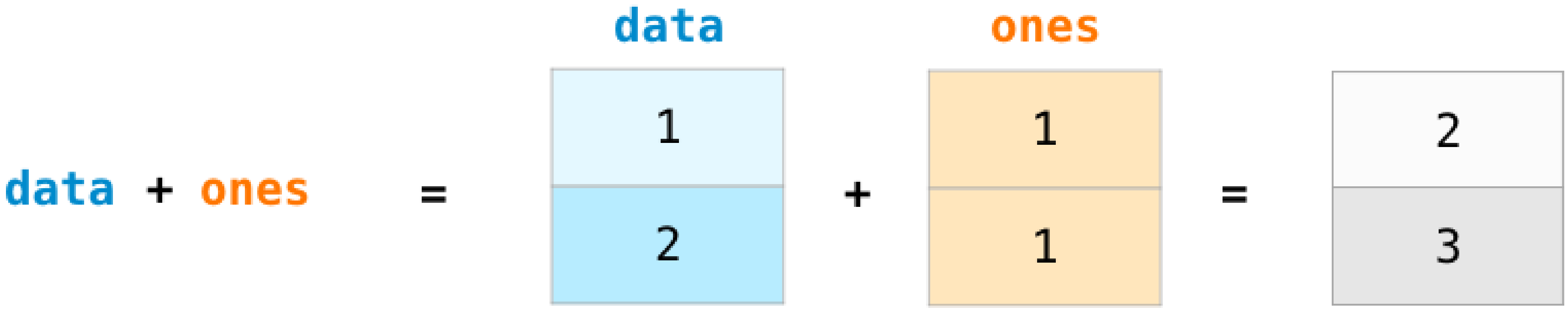



```
ones = np.ones(2)
```



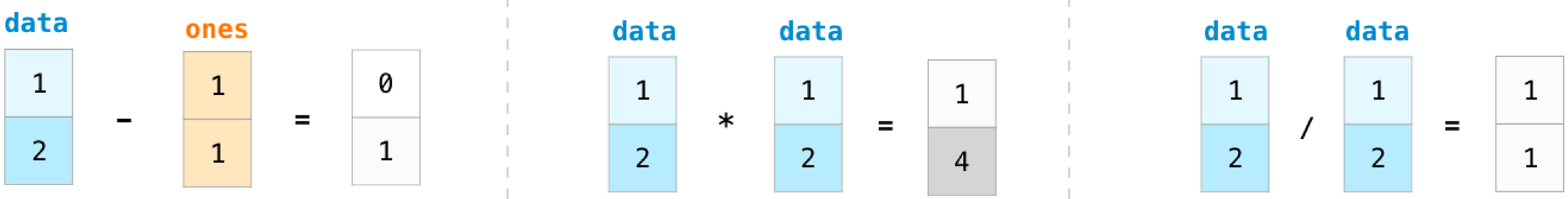
Array Arithmetic

Vector Addition (Parallel Processing)



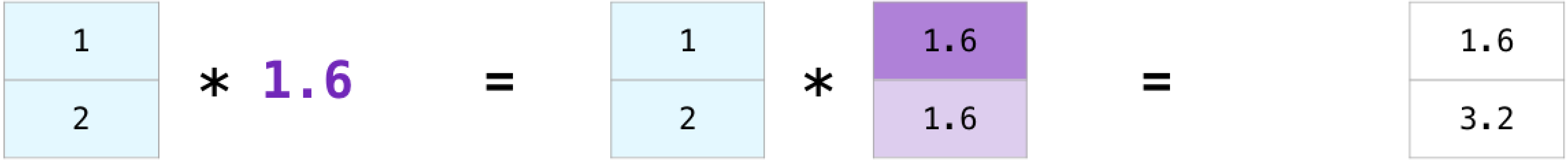
Array Arithmetic

Vector Arithmetic (Parallel Processing)



Array Arithmetic

Scalar-Vector Arithmetic (Parallel Processing)



Array Creation

Demo Program: array2_arithmetic.py

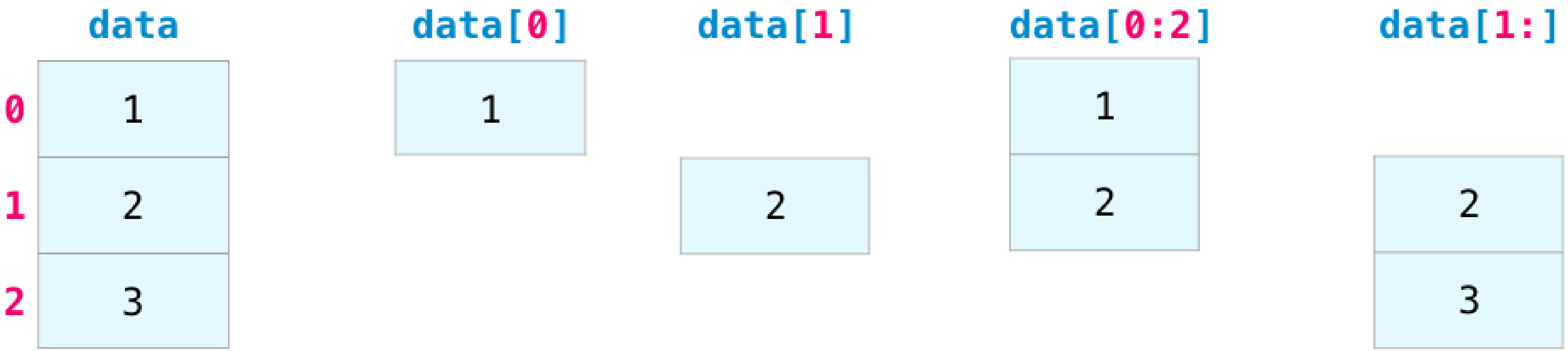
```
import numpy as np
data = np.array([1, 2])
ones = np.ones(2)
identity = np.eye(2)
zeros = np.zeros(2)
half = ones * 0.5
print("data: ", data)
print("ones: ", ones)
print("I: ", identity)
print("zeros: ", zeros)
print("Add: ", (data + ones))
print("Sub: ", (data - ones))
print("Mul: ", (data * half))
print("Div: ", (data + half))
print("Scalar-Vector: ", (0.6*data))
```

```
data:  [1 2]
ones:  [1. 1.]
I:  [[1. 0.]
    [0. 1.]]
zeros:  [0. 0.]
Add:  [2. 3.]
Sub:  [0. 1.]
Mul:  [0.5 1. ]
Div:  [1.5 2.5]
Scalar-Vector:  [0.6 1.2]
```

Indexing and Slicing

SECTION 8.3

Indexing



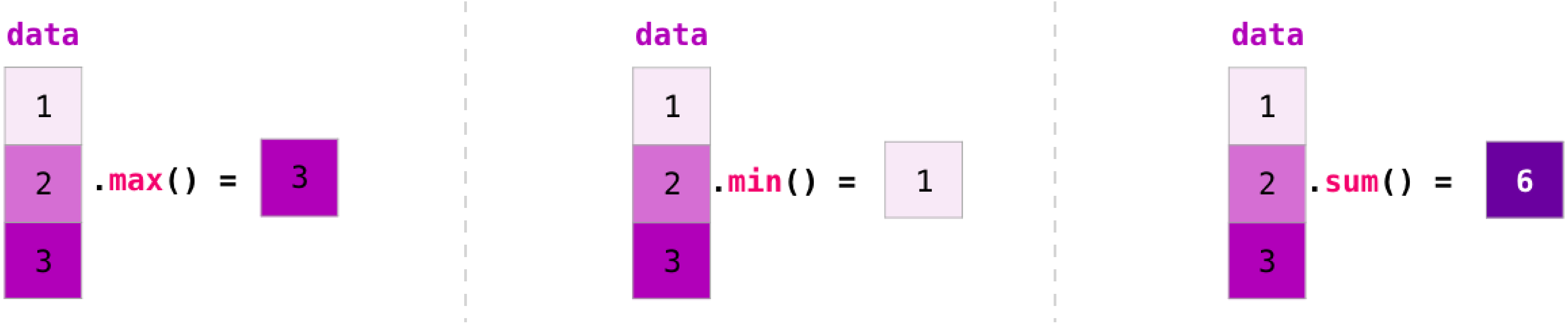
Array Creation

Demo Program: array3_index.py

```python
import numpy as np
data = np.arange(0, 11)  # 0-10
print("data      : ", data)
print("data[1]  : ", data[1])
print("data[-3]: ", data[-3])
print("data[2:5] : ", data[2:5])
print("data[-1:-9:-2] : ", data[-1:-9:-2])
print("data[-1:0] : ", data[-1:0])
print("data[-1:0:-1] : ", data[-1:0:-1])
print("data[3:-2] : ", data[3:-2])
print("data[:-2] : ", data[:-2])
print("data[2:] : ", data[2:])
```

```
data   : [0  1  2  3  4  5  6  7  8  9 10]
data[1] :  1
data[-3]:  8
data[2:5] :  [2 3 4]
data[-1:-9:-2] :  [10  8  6  4]
data[-1:0] :  []
data[-1:0:-1] :  [10  9  8  7  6  5  4  3  2  1]
data[3:-2] :  [3 4 5 6 7 8]
data[:-2] :  [0 1 2 3 4 5 6 7 8]
data[2:] :  [ 2  3  4  5  6  7  8  9 10]
```

Aggregates



Array Creation

Demo Program: array3_aggregates.py

```python
import numpy as np
data = np.arange(0, 11)  # 0-10
print("data       : ", data)
print("data.sum(): ", data.sum())
print("data.max(): ", data.max())
print("data.min(): ", data.min())
print("np.average(a): ", np.average(data))
print("np.mean(a): ", np.mean(data))
print("np.median(a): ", np.median(data))
print("np.std(a): ", np.std(data))
print("np.var(a): ", np.var(data))
```

```
data    : [0 1 2 3 4 5 6 7 8 9 10]
data.sum(): 55
data.max(): 10
data.min(): 0
np.average(a): 5.0
np.mean(a): 5.0
np.median(a): 5.0
np.std(a): 3.1622776601683795
np.var(a): 10.0
```

2D Array As Matrix/Table

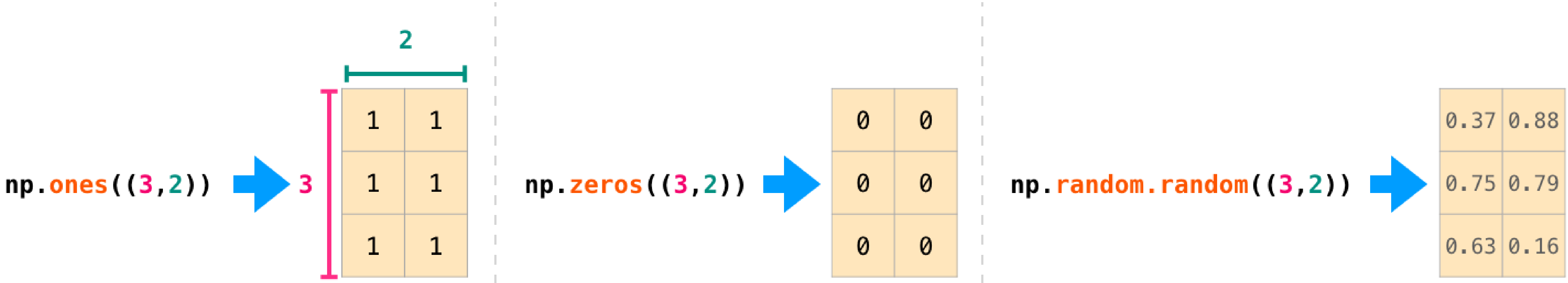
SECTION 8.4

Creating Matrices

```
np.array([[1,2],[3,4]])
```

1	2
3	4

2D Array Creation



Matrix Creation

Demo Program: array4_2D_create.py

```
import numpy as np
m1 = np.array([[1, 2], [3, 4]])
print("m1: ", m1)
m2 = np.array(np.matrix("5 6; 7 8"))
print("m2: ", m2)
m3 = np.zeros((2, 2))
print("m3: ", m3)
m4 = np.ones((2, 2))
print("m4: ", m4)
m5 = np.eye(2)
print("m5: ", m5)
m6 = np.random.randint(10, size=(2, 2))
print("m6: ", m6)
m7 = np.random.random((2, 2))
print("m7: ", m7)
```

```
m1:  [[1 2]
      [3 4]]
m2:  [[5 6]
      [7 8]]
m3:  [[0. 0.]
      [0. 0.]]
m4:  [[1. 1.]
      [1. 1.]]
m5:  [[1. 0.]
      [0. 1.]]
m6:  [[7 9]
      [5 2]]
m7:  [[0.92441049 0.79840959]
      [0.057992   0.98685019]]
```

Matrix Arithmetic

Matrix Addition (Parallel Processing)

$$\texttt{data} + \texttt{ones} = \overset{\texttt{data}}{\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}} + \overset{\texttt{ones}}{\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$$

Matrix Arithmetic

Scalar-Vector-Matrix Addition (Parallel Processing)

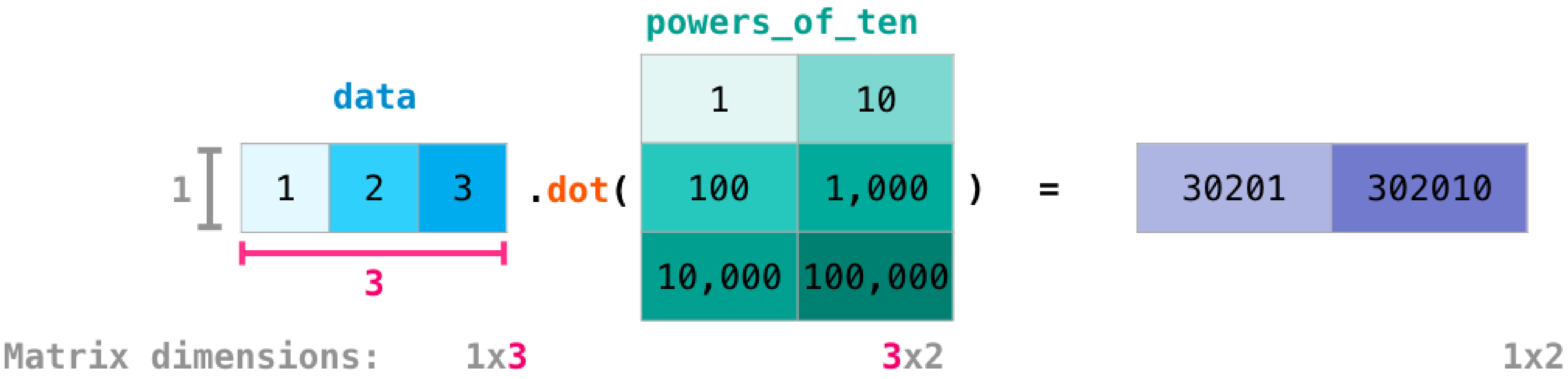
$$\texttt{data} + \texttt{ones\_row} = \underbrace{\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}}_{\texttt{data}} + \underbrace{\begin{bmatrix} 1 & 1 \end{bmatrix}}_{\texttt{ones\_row}} = \underbrace{\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}}_{\texttt{data}} + \underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}}_{\texttt{ones\_row}} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix}$$

Matrix/Vector/Scalar Operations

Demo Program: array4_2D_Operations.py

```
import numpy as np
data = np.array([
 [1, 2],
 [3, 4],
 [5, 6]
])
print(data)
ones32 = np.ones((3, 2))
ones = np.ones(2)
one  = 1
print("data+ones32:\n", (data+ones32))
print("data+ones:\n", (data+ones))
print("data+one:\n", (data+one))
```

Dot Product


powers_of_ten
data
1
1
2
3
.dot(
1
10
100
1,000
10,000
100,000
)
=
30201
302010
3
Matrix dimensions:
1x3
3x2
1x2

Dot Product

Demo Program: array4_dot.py

```
import numpy as np

data = np.array([1, 2, 3])
power = [10**i for i in range(6)]
power_of_ten = np.array(power).reshape((3, 2))
print("data:\n", data)
print("power:\n", power_of_ten)
dot_product = data.dot(power_of_ten)
print("dot_product:\n", dot_product)
```

```
data:
 [1 2 3]
power:
 [[     1     10]
 [   100   1000]
 [ 10000 100000]]
dot_product:
 [ 30201 302010]
```

Sum

sum([1, 100, 10,000] * [1, 2, 3])

sum([10, 1,000, 100,000] * [1, 2, 3])

1x2

1*1 + 2*100 + 3*10,000

1*10 + 2*1,000 + 3*100,000

= [30201, 302010]

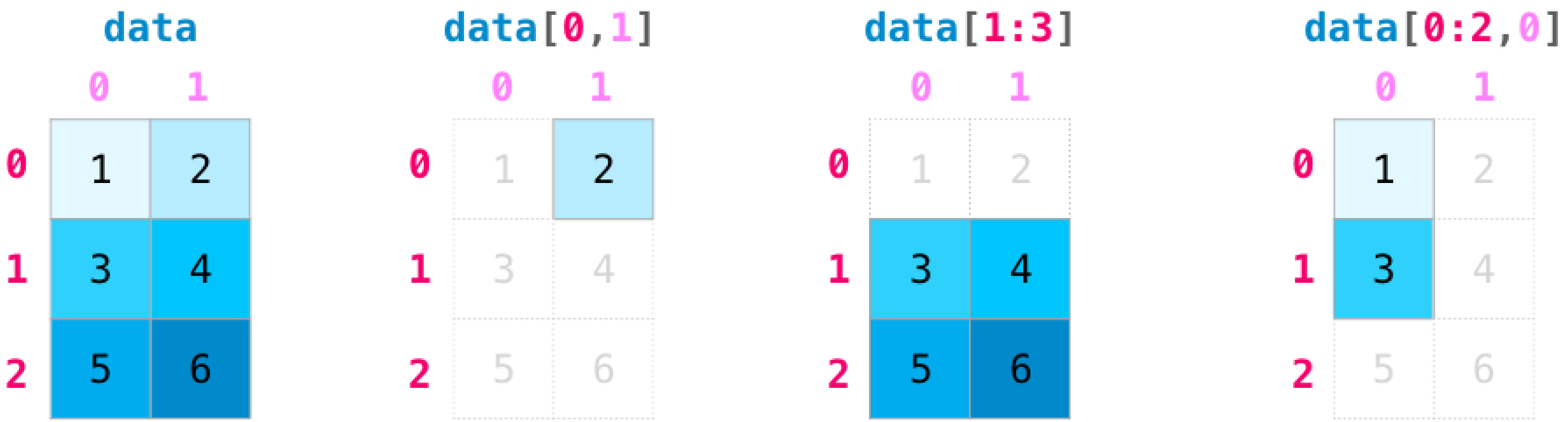
Dot Product

Demo Program: array4_dot2.py

```
import numpy as np
data = np.array([1, 2, 3])
power = [10**i for i in range(6)]
power_of_ten = np.array(power).reshape((3, 2))
c0 = power_of_ten[:, 0]
c1 = power_of_ten[:, 1]
print("data:", data)
print("c0:", c0)
print("c1:", c1)
x0 = data.dot(c0)
x1 = data.dot(c1)
print("sum of data * c0 : ", x0)
print("sum of data * c1 : ", x1)
s0 = sum(data * c0)
s1 = sum(data * c1)
print("sum of data * c0.T : ", s0)
print("sum of data * c1.T : ", s1)
```

```
data: [1 2 3]
c0: [     1   100 10000]
c1: [    10   1000 100000]
sum of data * c0 :  30201
sum of data * c1 :  302010
sum of data * c0.T :  30201
sum of data * c1.T :  302010
```

Matrix Indexing



Matrix Aggregation

`data`

$$\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix}\texttt{.max() = } \boxed{6}$$

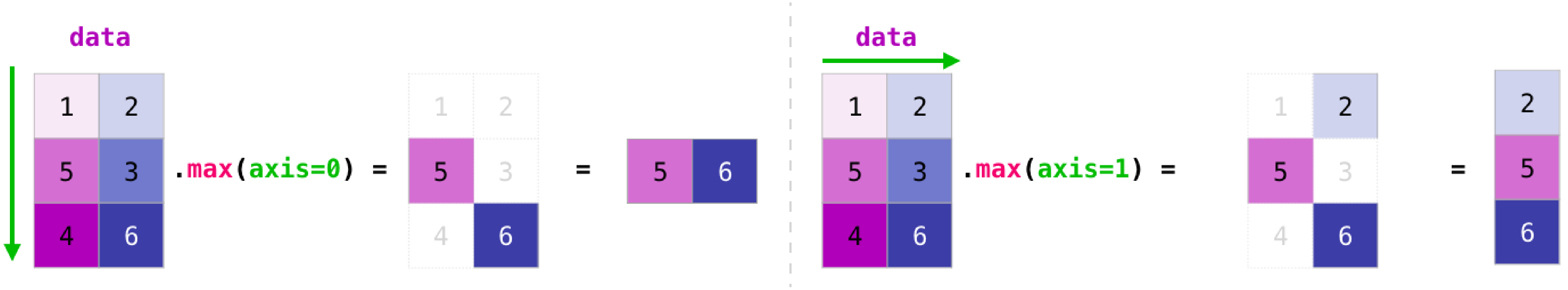
`data`

$$\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix}\texttt{.min() = } \boxed{1}$$

`data`

$$\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix}\texttt{.sum() = } \boxed{21}$$

Matrix Aggregation (Row/Column Level)



Transpose and Reshape

SECTION 8.5

Transposing and Reshaping

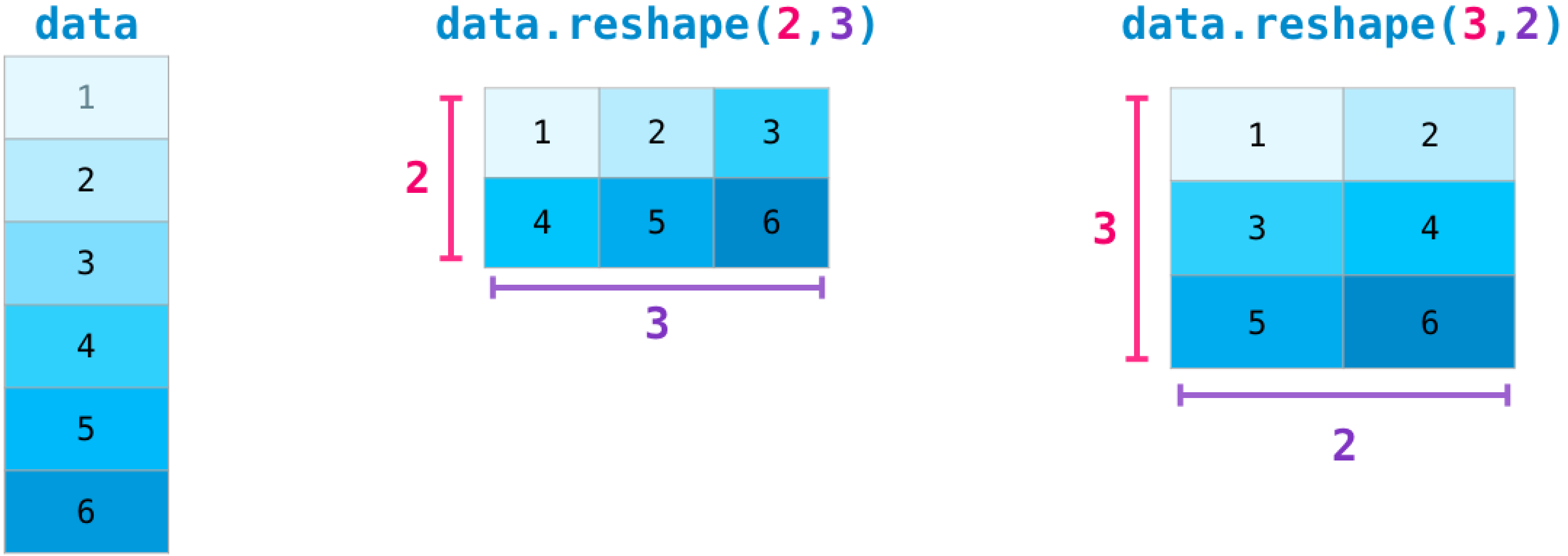
`data`

1	2
3	4
5	6

`data.T`

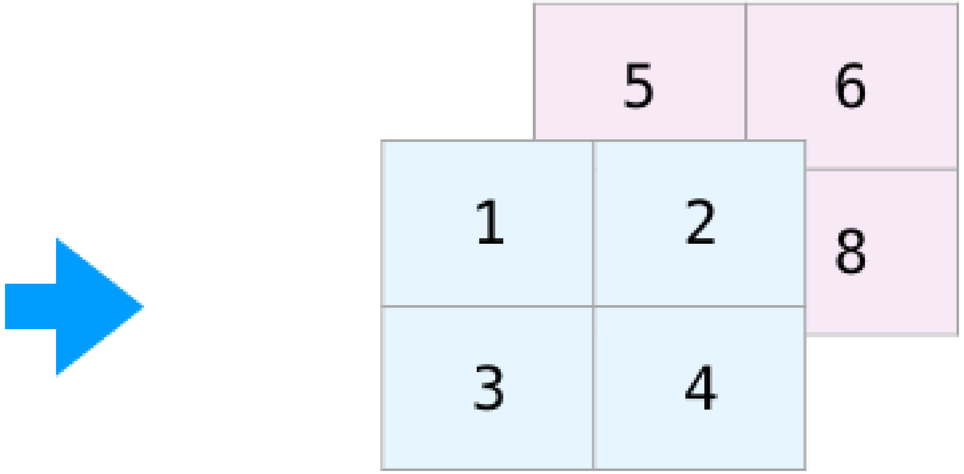
1	3	5
2	4	6

Transposing and Reshaping



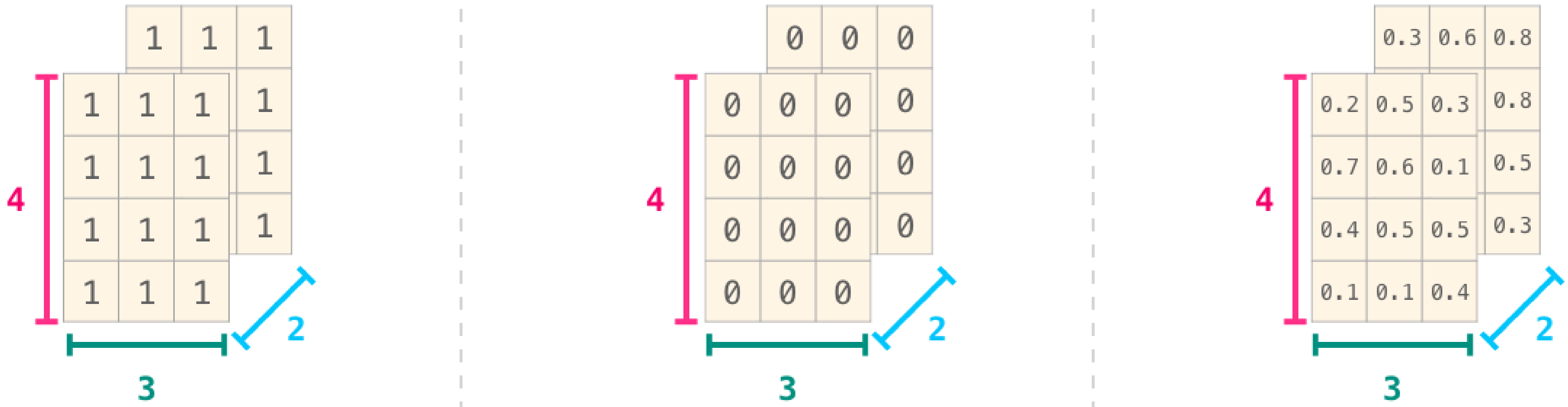
Multiple Dimension

```
np.array([ [[1,2],[3,4]],
           [[5,6],[7,8]] ])
```



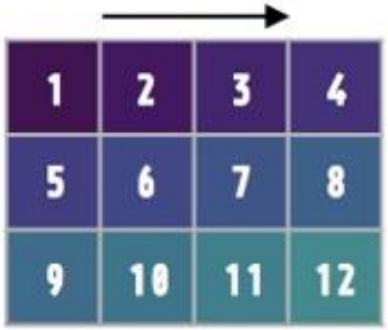
MD Array Creation

`np.ones((4,3,2))` `np.zeros((4,3,2))` `np.random.random((4,3,2))`



reshape & ravel

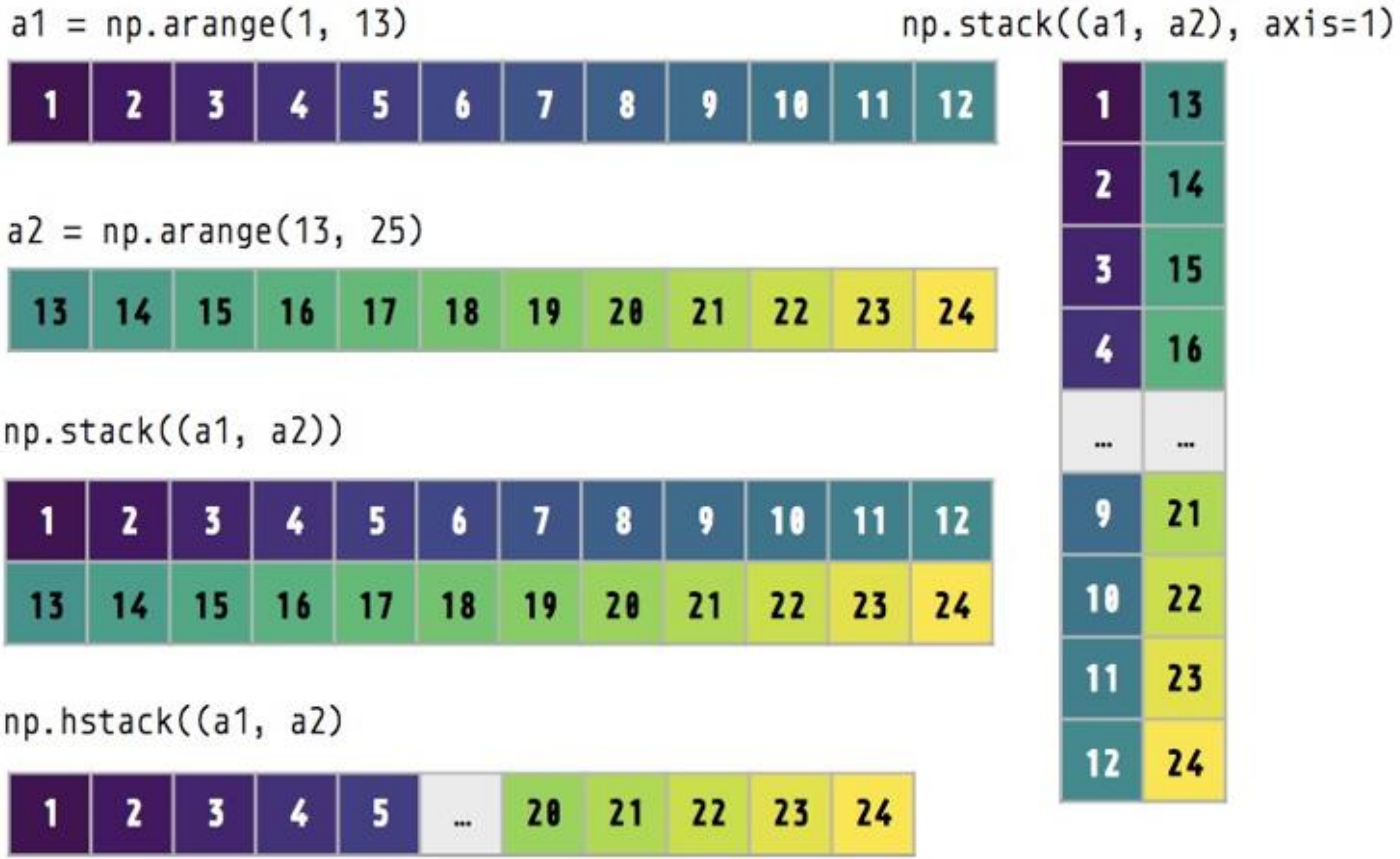
```
a1 = np.arange(1, 13)
```




```
a1.reshape(3, 4)
a1.reshape(-1, 4)
a1.reshape(3, -1)
  .ravel()  # back to 1D
```


```
a1.reshape(3, -1, order='F')
  .ravel(order='F')  # back to 1D
```

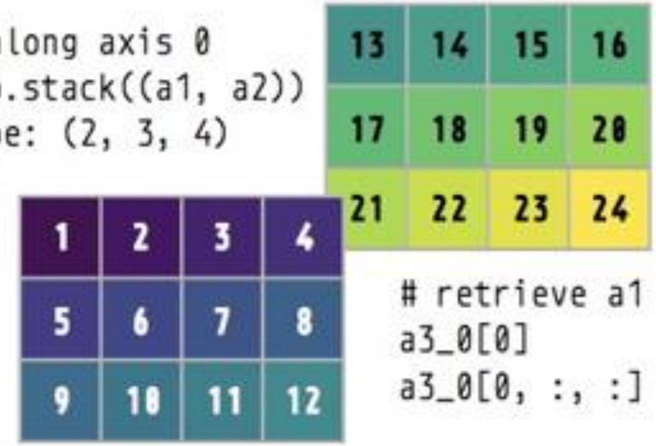
stack


a1 = np.arange(1, 13)
a2 = np.arange(13, 25)
np.stack((a1, a2))
np.hstack((a1, a2)
np.stack((a1, a2), axis=1)

3D array from 2D arrays

```
a1 = np.arange(1, 13).reshape(3, 4)
a2 = np.arange(13, 25).reshape(3, -1)
```

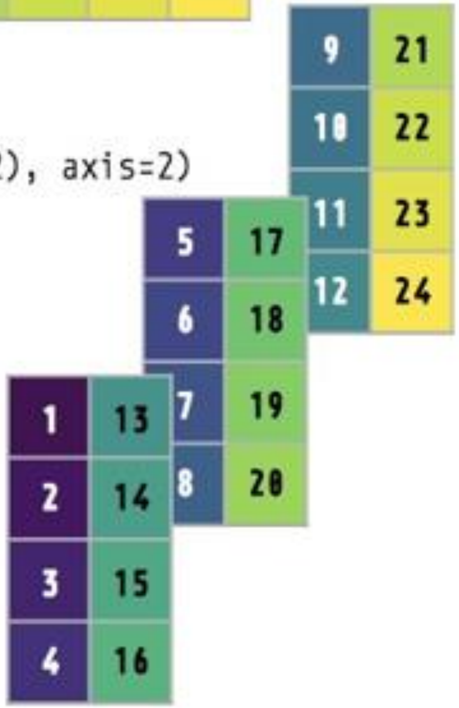

```
# stack along axis 0
a3_0 = np.stack((a1, a2))
a3_0.shape: (2, 3, 4)
```



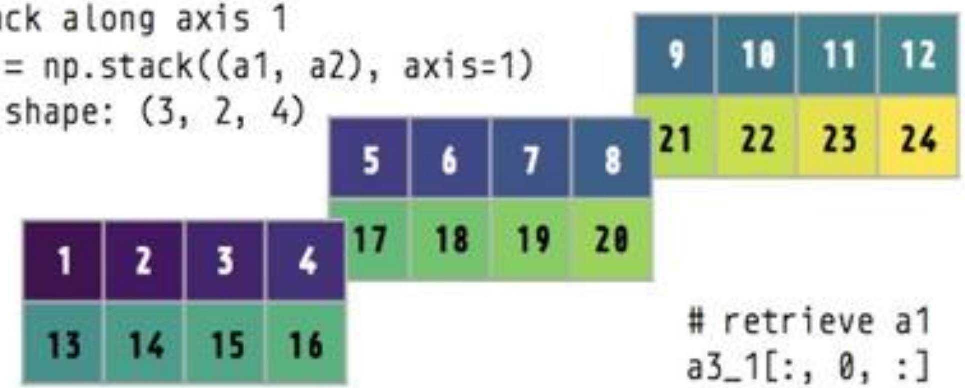

```
# retrieve a1
a3_0[0]
a3_0[0, :, :]
```

```
# stack along axis 2
a3_2 = np.stack((a1, a2), axis=2)
a3_2.shape: (3, 4, 2)
```

```
# retrieve a1
a3_2[:, :, 0]
```

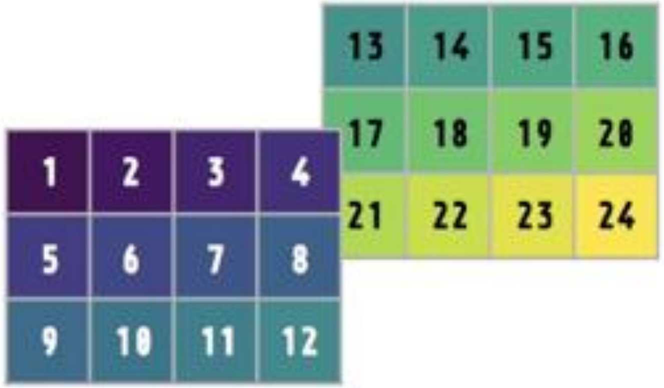



```
# stack along axis 1
a3_1 = np.stack((a1, a2), axis=1)
a3_1.shape: (3, 2, 4)
```




```
# retrieve a1
a3_1[:, 0, :]
```

flatten 3D array

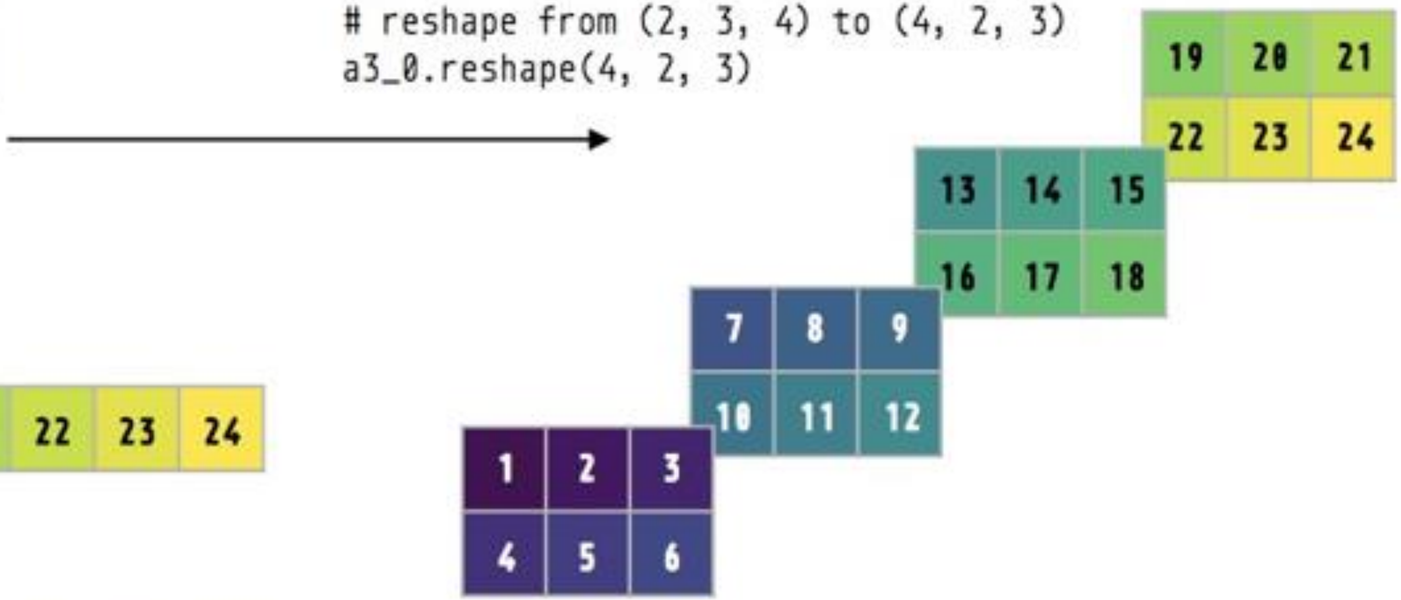



```
# flatten/ravel
a3_0.ravel()
```


```
# flatten/ravel
a3_0.ravel(order='F')
```

reshape 3D array

```
# reshape from (2, 3, 4) to (4, 2, 3)
a3_0.reshape(4, 2, 3)
```



Insertion/Deletion of Row and Column

SECTION 8.6

Array Creation

Demo Program: array5_InsDel1.py

```
import numpy as np

a = np.array([['Mon', 18, 20, 22, 17], ['Tue', 11, 18, 21, 18],
              ['Wed', 15, 21, 20, 19], ['Thu', 11, 20, 22, 21],
              ['Fri', 18, 17, 23, 22], ['Sat', 12, 22, 20, 18],
              ['Sun', 13, 15, 19, 16]])

m = np.reshape(a, (7, 5))
print(m)
```

Array Creation

Demo Program: array5_InsDel1.py

```
[['Mon' '18' '20' '22' '17']
 ['Tue' '11' '18' '21' '18']
 ['Wed' '15' '21' '20' '19']
 ['Thu' '11' '20' '22' '21']
 ['Fri' '18' '17' '23' '22']
 ['Sat' '12' '22' '20' '18']
 ['Sun' '13' '15' '19' '16']]
```

Add a Row

Demo Program: array5_InsDel2.py

```
import numpy as np

m = np.array([['Mon', 18, 20, 22, 17], ['Tue', 11, 18, 21, 18],
              ['Wed', 15, 21, 20, 19], ['Thu', 11, 20, 22, 21],
              ['Fri', 18, 17, 23, 22], ['Sat', 12, 22, 20, 18],
              ['Sun', 13, 15, 19, 16]])

m_r = np.append(m, [['Avg', 12, 15, 13, 11]], 0)

print(m_r)
```

Add a Row

Demo Program: array5_InsDel2.py

```
[['Mon' '18' '20' '22' '17']
 ['Tue' '11' '18' '21' '18']
 ['Wed' '15' '21' '20' '19']
 ['Thu' '11' '20' '22' '21']
 ['Fri' '18' '17' '23' '22']
 ['Sat' '12' '22' '20' '18']
 ['Sun' '13' '15' '19' '16']
 ['Avg' '12' '15' '13' '11’]]
```

Add a Column

Demo Program: array5_InsDel3.py

```
import numpy as np

m = np.array([['Mon', 18, 20, 22, 17], ['Tue', 11, 18, 21, 18],
              ['Wed', 15, 21, 20, 19], ['Thu', 11, 20, 22, 21],
              ['Fri', 18, 17, 23, 22], ['Sat', 12, 22, 20, 18],
              ['Sun', 13, 15, 19, 16]])

m_c = np.insert(m, [5], [[1], [2], [3], [4], [5], [6], [7]], 1)

print(m_c)
```

Add a Column

Demo Program: array5_InsDel3.py

```
[['Mon' '18' '20' '22' '17' '1']
 ['Tue' '11' '18' '21' '18' '2']
 ['Wed' '15' '21' '20' '19' '3']
 ['Thu' '11' '20' '22' '21' '4']
 ['Fri' '18' '17' '23' '22' '5']
 ['Sat' '12' '22' '20' '18' '6']
 ['Sun' '13' '15' '19' '16' '7']]
```

Delete a Row

Demo Program: array5_InsDel4.py

```
import numpy as np

m = np.array([['Mon', 18, 20, 22, 17], ['Tue', 11, 18, 21, 18],
              ['Wed', 15, 21, 20, 19], ['Thu', 11, 20, 22, 21],
              ['Fri', 18, 17, 23, 22], ['Sat', 12, 22, 20, 18],
              ['Sun', 13, 15, 19, 16]])

m = np.delete(m, [2], 0)

print(m)
```

Delete a Row

Demo Program: array5_InsDel4.py

```
[['Mon' '18' '20' '22' '17']
 ['Tue' '11' '18' '21' '18']
 ['Thu' '11' '20' '22' '21']
 ['Fri' '18' '17' '23' '22']
 ['Sat' '12' '22' '20' '18']
 ['Sun' '13' '15' '19' '16’]]
```

Delete a Column

Demo Program: array5_InsDel5.py

```
import numpy as np

m = np.array([['Mon', 18, 20, 22, 17], ['Tue', 11, 18, 21, 18],
              ['Wed', 15, 21, 20, 19], ['Thu', 11, 20, 22, 21],
              ['Fri', 18, 17, 23, 22], ['Sat', 12, 22, 20, 18],
              ['Sun', 13, 15, 19, 16]])

m = np.delete(m, [2], 1)

print(m)
```

Delete a Column

Demo Program: array5_InsDel5.py

```
[['Mon' '18' '22' '17']
 ['Tue' '11' '21' '18']
 ['Wed' '15' '20' '19']
 ['Thu' '11' '22' '21']
 ['Fri' '18' '23' '22']
 ['Sat' '12' '20' '18']
 ['Sun' '13' '19' '16']]
```

Modify a Row

Demo Program: array5_InsDel6.py

```
import numpy as np

m = np.array([['Mon', 18, 20, 22, 17], ['Tue', 11, 18, 21, 18],
              ['Wed', 15, 21, 20, 19], ['Thu', 11, 20, 22, 21],
              ['Fri', 18, 17, 23, 22], ['Sat', 12, 22, 20, 18],
              ['Sun', 13, 15, 19, 16]])

m[3] = ['Thu', 0, 0, 0, 0]

print(m)
```

Modify a Row

Demo Program: array5_InsDel6.py

```
[['Mon' '18' '20' '22' '17']
 ['Tue' '11' '18' '21' '18']
 ['Wed' '15' '21' '20' '19']
 ['Thu' '0' '0' '0' '0']
 ['Fri' '18' '17' '23' '22']
 ['Sat' '12' '22' '20' '18']
 ['Sun' '13' '15' '19' '16’]]
```

End of Chapter 2