Introduction to Bioinformatics using Python

Lecture 3: Python basic concepts

Dr. Alexey Larionov

28 October 2024

www.cranfield.ac.uk

Lecture plan (learning objectives)

At end of this lecture, you should be able to:

- Operate simple Python functions for command line input / output
- Understand Python variables and built-in data types
- Describe operations and functions for Boolean and numeric data types
- Understand strings indexing and perform operations with strings
- Control `print()` function

Lecture plan (learning objectives)

At end of this lecture, you should be able to:

- Operate simple Python functions for command line input / output
- Understand Python variables and built-in data types
- Describe operations and functions for Boolean and numeric data types
- Understand strings indexing and perform operations with strings
- Control `print()` function

Simple input and output

From the Hello Word example, you already know how to use **`print()`** function for output:

```
print("DNA bases: ATGC")
```

This can also be done using **variables**:

```
dna = "DNA bases: ATGC"
print(dna)
```

In interactive terminals and Jupyter sometime `print()` may be omitted … Don't omit it, unless you understand what you are doing.

Variable

You can also use **`input()`** function to save data to variables:

```
dna = input("Enter your input: ")
...
print(dna)
```

This will print prompt, wait for user input and save input to the variable

Lecture plan (learning outcomes)

At end of this lecture, you should be able to:

- Operate simple Python functions for command line input / output
- Understand Python variables and built-in data types
- Describe operations and functions for Boolean and numeric data types
- Understand strings indexing and perform operations with strings
- Control `print()` function

Variable

What happens when we assign (create) a variable?

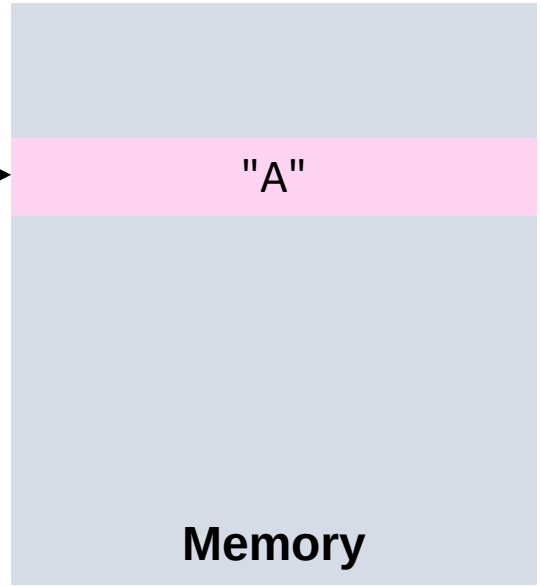
```
nucleotide = "A"
```

Assignment operator

A new memory object is created and the name `'nucleotide'` pointing to it

Somewhere under the hood in Python

```
nucleotide
2201285084400
<class 'str'>
```




```
nucleotide = "A"

id(nucleotide)
type(nucleotide)
print(nucleotide)
```

Python commands

```
2201285084400
<class 'str'>
A
```

Python outputs

In addition to **name** and **content**, variables have other properties including **address** in the memory and data **type**

https://realpython.com/python-variables

Data type

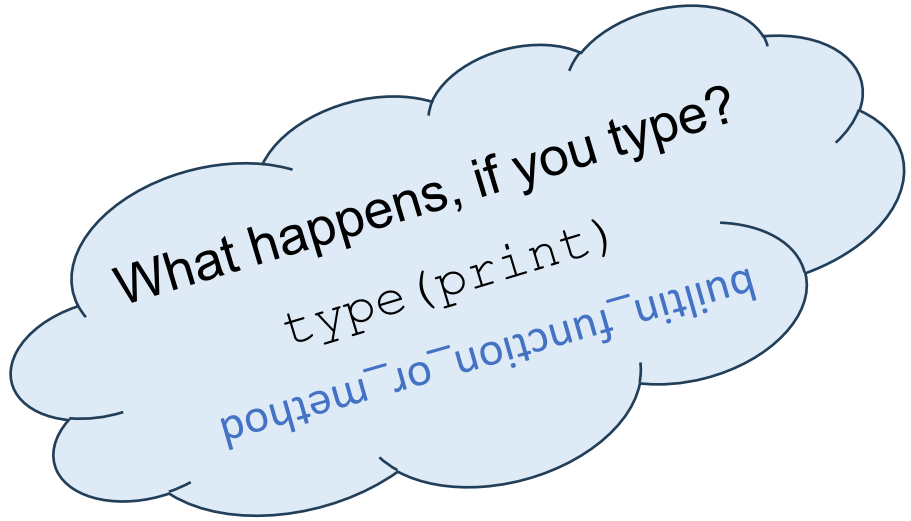
`type()` function reports the data type

```
type(dna)
<class 'str'>
```

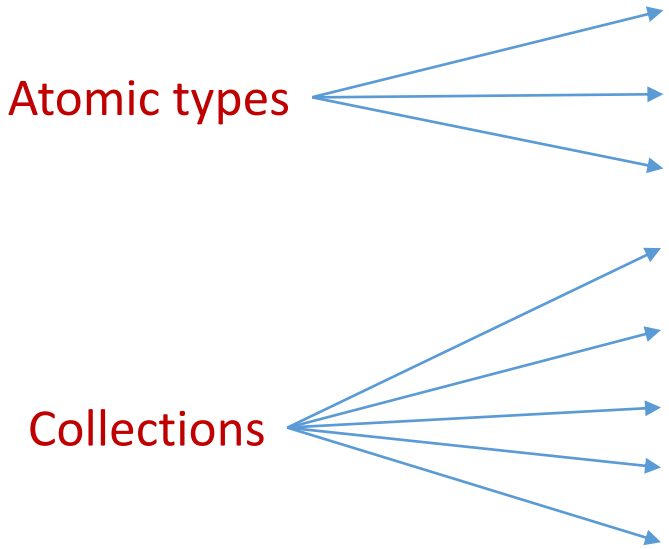
→ **str**ing of characters

```
type(42)
<class 'int'>
```

→ **int**eger



Main built-in data types in Python



Class	Description
bool	Boolean value
int	Integer (of any size!)
float	Floating-point number
str	Character string
tuple	*Immutable* sequence of objects
list	*Mutable* sequence of objects
set	Unordered collection of *distinct* objects
dict	Key-Value pairs (like dictionary)

There are also other data types (e.g. frozenset, complex numbers, ranges etc)
https://docs.python.org/3/library/stdtypes.html

Dynamic assignment of variable types

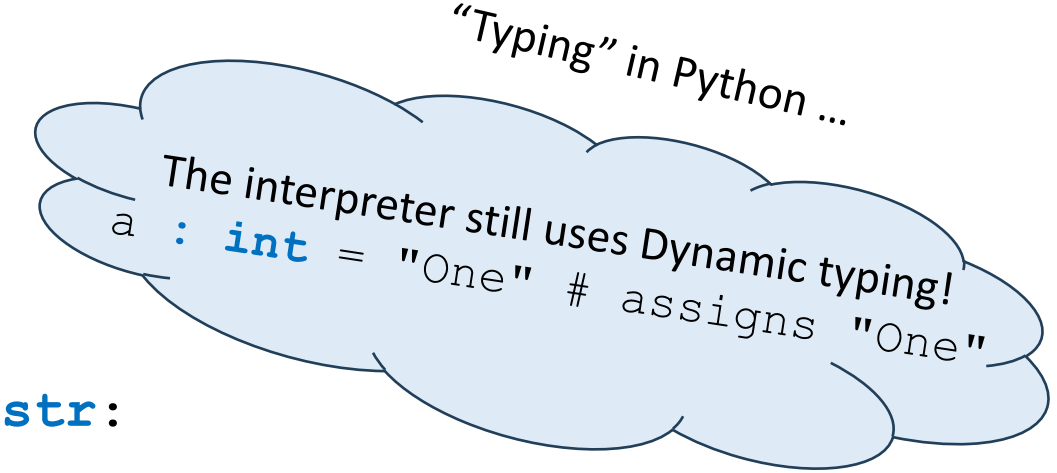
In Python the type of variable is guessed by the interpreter from the context

```
a = 1      # saved as integer
a = 1.0    # saved as "float"
a = "One" # saved as string
```

Since Python 3.6 there is an option for ***type hints***

(can be used with 3rd-party ***type-checkers***, *Cyton*, etc)

```
a : int   = 1
a : float = 1.0
a : str = "One"
def my_function(argument: str) -> str:
    # Do something with the strings …
```



https://medium.com/@ageitgey/learn-how-to-use-static-type-checking-in-python-3-6-in-10-minutes-12c86d72677b

https://realpython.com/python-type-checking

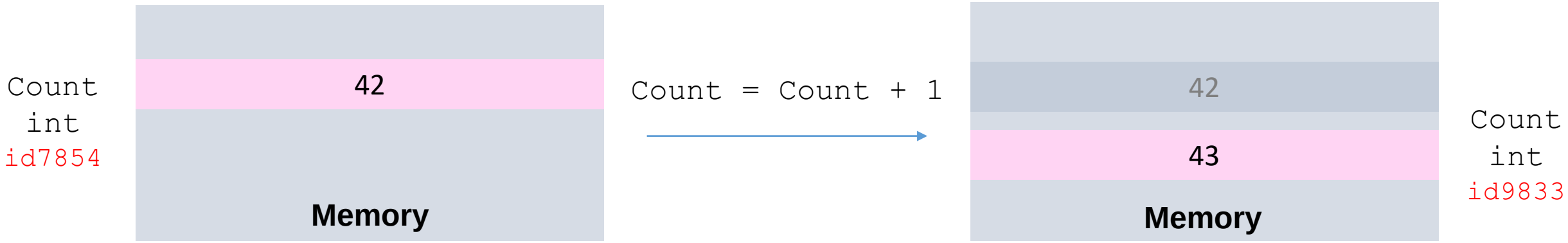
Mutability: changing object in place

Changing in place = at the same memory location

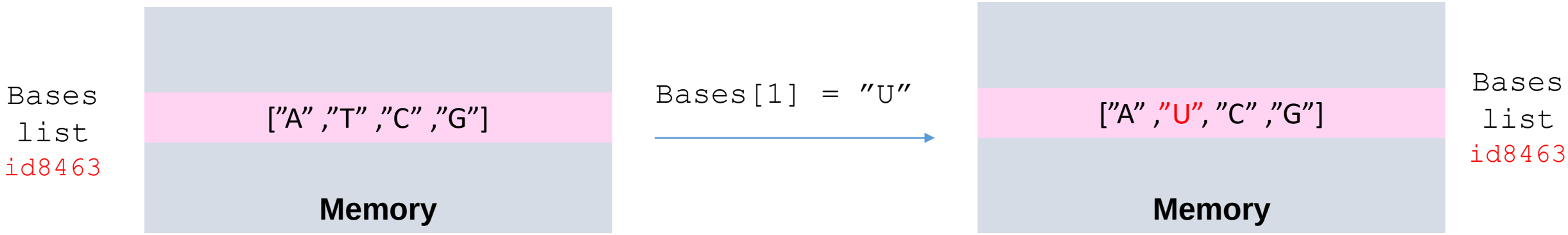
Class	Description	Immutable	Mutable
bool	Boolean value	✓	
int	Integer (of any size!)	✓	
float	Floating-point number	✓	
str	Character string	✓	
tuple	Immutable sequence of objects	✓	
list	Mutable sequence of objects		✓
set	Unordered set of distinct objects		✓
dict	Key-Value pairs (like dictionary)		✓

Mutability: changing object in place

Immutable: a new memory object is created if we change it



Mutable: the memory object is modified in place



Variables *vs* Memory objects

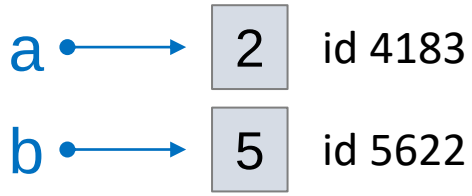
1

a = 2

2

a = 2

b = 5

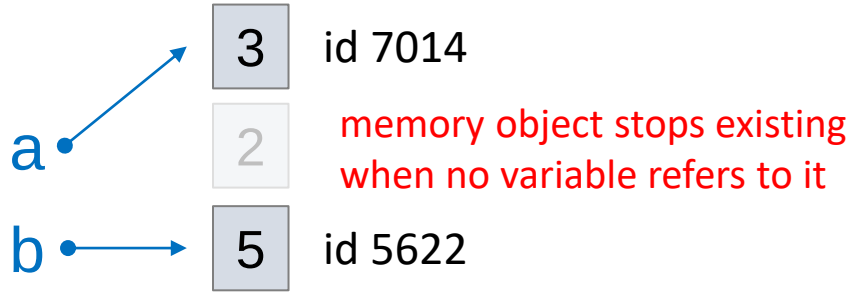


3

a = 2

b = 5

a = 3



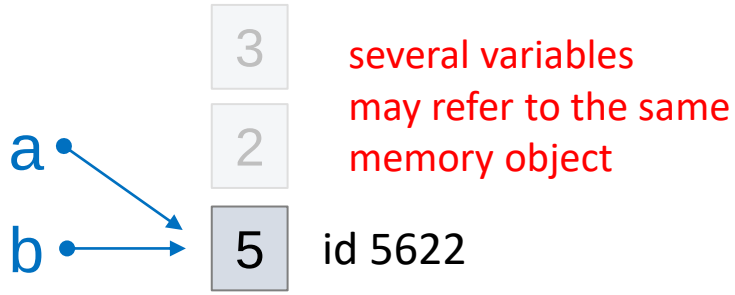
4

a = 2

b = 5

a = 3

a = b



5

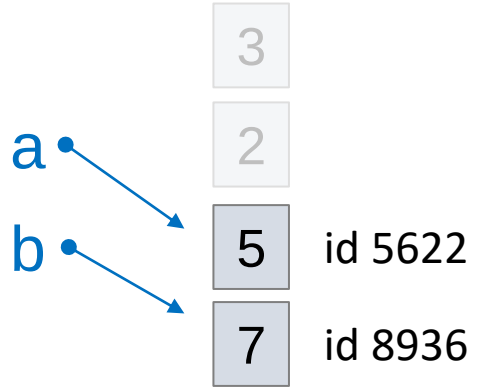
a = 2

b = 5

a = 3

a = b

b = 7



Names of variables

You can choose the name you like but remember:

- Variable names are case sensitive
- Can only contain A-Z, a-z, 0-9 or _
- Cannot start with a number
- Should not be one of the reserved words

Also: don’t use dots in names !

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

Naming conventions

The Python community has naming conventions

- joined_lower for variables, functions, methods and attributes
- ALL_CAPS for constants
- StudlyCaps for classes (“Camel” style)
- Underscores: _internal vs accessible attributes/methods in classes
  (some fancy _ _ double-underscores _ _ in special cases as part of syntax)

https://pep8.org

Lecture plan (learning outcomes)

At end of this lecture, you should be able to:

- ✓ Operate simple Python functions for command line input / output
- ✓ Understand Python variables and built-in data types
- Describe operations and functions for Boolean and numeric data types
- Understand strings indexing and perform operations with strings
- Control `print()` function

Booleans

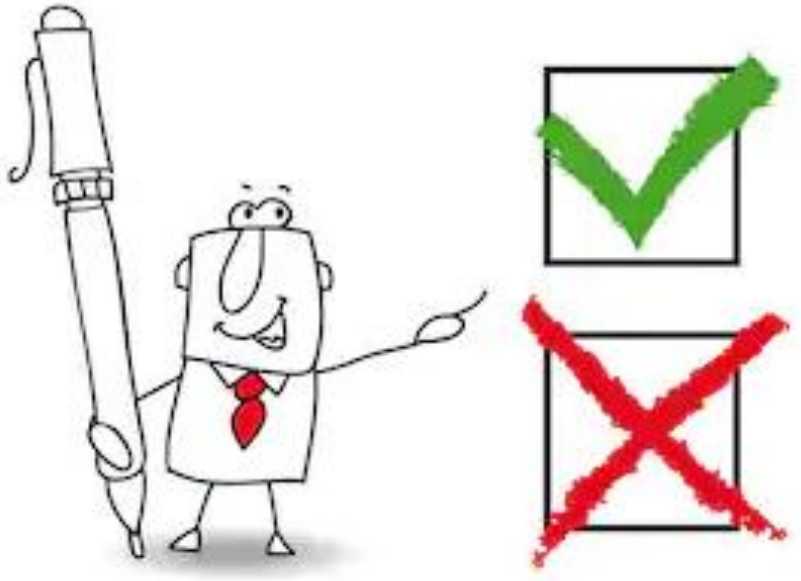
Booleans are binary values: True or False

Boolean algebra rules

```
T = True
F = False

print("not T: ", not T)        # False
print("not F: ", not F)        # True

print("T and  F: ", T and F)   # False
print("T and  T: ", T & T)     # True
print("F and  F: ", F & F)     # False

print("T or  F: ", T or F)     # True
print("T or  T: ", T | T)      # True
print("F or  F: ", F | F)      # False
```

```
& = and
| = or
```

George Boole
The Laws of Thought
1854



Integers

$\mathbb{Z}\{\ldots -2, -1, 0, 1, 2, 3, 4 \ldots\}$

In Python integers are of unlimited size !!!
(well… limited by computer memory :)

```
# addition
x + y

# subtraction
x - y

# multiplication
x * y

# normal division (returns float)
x / y

# "floor division"
# "integer division"
x // y

# integer remainder
x % y

# power
x ** y
```

Not x ^ y

Try this:

x = 9 (or -9)
y = 4

x // y ?
x % y ?

Floats

- In Python real numbers are numbers with floating points (floats)

1.0
0.33333333
3.141592
2.7182818

```
# addition
x + y

# subtraction
x - y

# multiplication
x * y

# normal division
x / y

# "floor division"
# "integer division"
x // y

# integer remainder
x % y

# power
x ** y
```

Imprecision in floats

Floats are not exact

Try this:

```
import numpy as np
np.arange(0, 1, 0.1)
np.arange(0, 1, 0.1).tolist()
np.arange(0, 1, 0.1).round(1) .tolist()
```

```
1.0 / 3.0
0.3333333333333333
10.0 / 3.0
3.3333333333333335
```

```
0.1 + 0.1
0.2

0.1 + 0.1 + 0.1
0.30000000000000004
```

```
4294967296.0**2
1.8446744073709552e+19

4294967296**2
18446744073709551616
```

Difference ~ 384

Explore more at:
https://docs.python.org/3/tutorial/floatingpoint.html

Advanced use of assignment operator

"Abbreviated assignments"

Addition assignment
Decrement assignment
Multiplication assignment
Division assignment
…

Operator	Example	Equivalent to
=	x = 5	x = 5
	x = y = 5	x = 5 y = 5
	x , y , z = 5, 6, 7	x = 5 y = 6 z = 7
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

!?

Bitwise XOR

Shift operators

Fancy figures of speech …

a = 5, 6, 7

x, y, z = a

_, y, _ = (5, 6, 7)

Shift operators

>> shift to the right

8 | 0 | 0 | 1 | 0 | 0 | 0 |

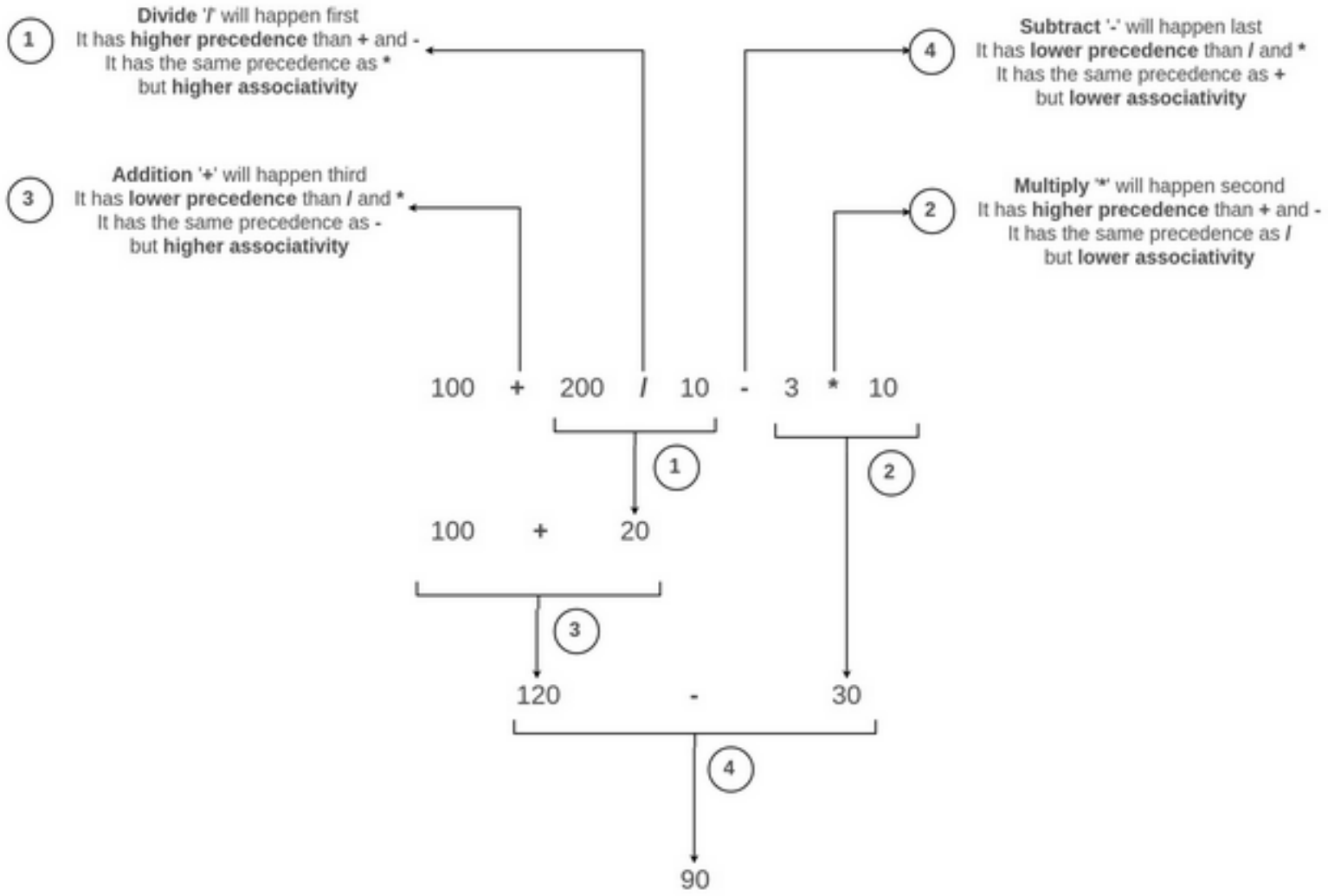
by 3 positions →

8 >> 3 is 1 | 0 | 0 | 0 | 0 | 0 | 1 |

Decimal Binary

Operator precedence and associativity

Operator precedence and associativity in Python is the same as in conventional math



/ and *
both have the same precedence
but Left to Right (**LTR**) associativity

+ and -
both have the same precedence
but Left to Right (**LTR**) associativity

/ and *
have the higher precedence
than + and -

math() module

`math()` module can be used for various mathematical and scientific operations

Some example of functions available in the module:

```
import math


math.ceil(10.12)  # return the ceiling of a number (upper integer)
math.floor(10.66) # return the floor of a number (lower integer)
math.exp(5)       # return e raised to the power
math.sqrt(81)     # return square root of a number
math.cos(0.05)    # return the cosine of a radian angle
math.pi           # mathematical constant pi
```

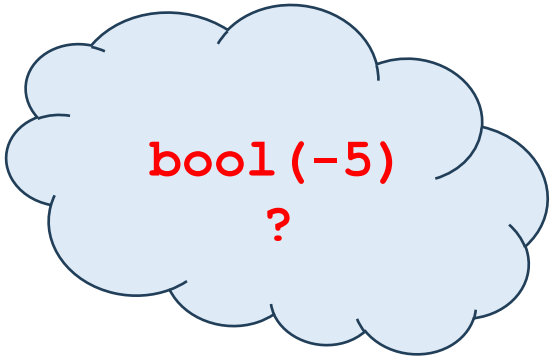
For full list of math functions: https://docs.python.org/3/library/math.html

Types conversion (type “casting”)

There are several functions can convert one data type to another

`int()`, **`float()`**, **`str()`**, and **`bool()`**
convert to **integer**, **floating** point, **string** and **Boolean** types, respectively

Examples:	Output:
`1.0/2.0`	`0.5`
`1/2`	`0`
`float(1)/float(2)`	`0.5`
`int(3.1415926)`	`3`
`str(3.1415926)`	`3.1415926`
`bool(1)`	`True`
`bool(0)`	**`False`**



Everything that isn’t False is casted to True

Lecture plan (learning outcomes)

At end of this lecture, you should be able to:

- ✓ Operate simple Python functions for command line input / output
- ✓ Understand Python variables and built-in data types
- ✓ Describe operations and functions for Boolean and numeric data types
- Understand strings indexing and perform operations with strings
- Control `print()` function

Strings

- Strings (the type name is `str`) are **immutable** objects used to handle text data
- Under the hood, strings are sequences of **Unicode** codes representing characters or formatting
- You can define strings in several ways:

```
string1 = "This is a string in double quotes"
string2 = 'This is a string in single quotes'

string3 = '''This is a string which can be
spread over several lines using single
quotes'''

string4 = """The same can be done
using double quotes : amazing! """
```

As far as Python syntax is concerned, there is no difference in **single** or **double quoted** string

Special characters

```
print("DNA bases: ATGC")

# Back slash \ is used to add special meaning

print("DNA bases:\tATGC")
print("DNA bases:\nATGC")
print("DNA bases:\rATGC")

print("DNA bases:\"ATGC\"")
print("DNA bases:\\ATGC")

# Explicit Unicode codes for a character

print("\u3452")
print(hex(ord("㑒")))

# And so on … … …
```

Try it yourself !

Tab
New line
Carriage return

Escape special meaning with one more back slash \

㑒
0x3452

String operations

Operator	Result	Meaning
len(str)	int	Return the length of the string
str + str	str	Concatenate two strings
str * int	str	Replicate the string
str in str	bool	Check if a string is present in another sting
str[int]	str	Read a character at specified index position
str[int:int]	str	Extract sub-string using indices

```
DNA1 = "ATTCG"
DNA2 = "GGATC"

# Joining strings
DNA3 = DNA1 + DNA2
```

Separator

List of stings

```
DNA3 = "".join([DNA1,DNA2])  <-->  "-".join([DNA1,DNA2])

tandem_repeat = DNA1 * 5
print(tandem_repeat + " has length: " + str(len(tandem_repeat)))
```

```
ATTCGATTCGATTCGATTCGATTCG has length: 25
```

```
# Searching in strings
motif1 = "TCGAT"
motif2 = "TCCT"
print(motif1 in tandem_repeat) # TRUE
print(motif2 in tandem_repeat) # FALSE
```

String operations

Operator	Result	Meaning
len(str)	int	Return the length of the string
str + str	str	Concatenate two strings
str * int	str	Replicate the string
str in str	bool	Check if a string is present in another sting
str[int]	str	Read a character at specified index position
str[int:int]	str	Extract sub-string using indices

```
DNA1 = "ATTCG"
DNA2 = "GGATC"

# Joining strings
DNA3 = DNA1 + DNA2
```

Separator — List of stings

```
DNA3 = "".join([DNA1,DNA2])  <-->  "-".join([DNA1,DNA2])

tandem_repeat = DNA1 * 5
print(tandem_repeat + " has length: " + str(len(tandem_repeat)))
```

```
ATTCGATTCGATTCGATTCGATTCG has length: 25
```

```
# Searching in strings
motif1 = "TCGAT"
motif2 = "TCCT"
print(motif1 in tandem_repeat) # TRUE
print(motif2 in tandem_repeat) # FALSE
```

String operations

Operator	Result	Meaning
len(str)	int	Return the length of the string
str + str	str	Concatenate two strings
str * int	str	Replicate the string
str in str	bool	Check if a string is present in another sting
str[int]	str	Read a character at specified index position
str[int:int]	str	Extract sub-string using indices

```
DNA1 = "ATTCG"
DNA2 = "GGATC"

# Joining strings
DNA3 = DNA1 + DNA2
```

Separator — List of stings

```
DNA3 = "".join([DNA1,DNA2])  <-->  "-".join([DNA1,DNA2])

tandem_repeat = DNA1 * 5
print(tandem_repeat + " has length: " + str(len(tandem_repeat)))
```

```
ATTCGATTCGATTCGATTCGATTCG has length: 25
```

```
# Searching in strings
motif1 = "TCGAT"
motif2 = "TCCT"
print(motif1 in tandem_repeat) # TRUE
print(motif2 in tandem_repeat) # FALSE
```

String operations

Operator	Result	Meaning
len(str)	int	Return the length of the string
str + str	str	Concatenate two strings
str * int	str	Replicate the string
str in str	bool	Check if a string is present in another sting
str[int]	str	Read a character at specified index position
str[int:int]	str	Extract sub-string using indices

```
DNA1 = "ATTCG"
DNA2 = "GGATC"

# Joining strings
DNA3 = DNA1 + DNA2
```

Separator — List of stings

```
DNA3 = "".join([DNA1,DNA2])  <-->  "-".join([DNA1,DNA2])

tandem_repeat = DNA1 * 5
print(tandem_repeat + " has length: " + str(len(tandem_repeat)))
```

```
ATTCGATTCGATTCGATTCGATTCG has length: 25
```

```
# Searching in strings
motif1 = "TCGAT"
motif2 = "TCCT"
print(motif1 in tandem_repeat) # TRUE
print(motif2 in tandem_repeat) # FALSE
```

String operations

Operator	Result	Meaning
len(str)	int	Return the length of the string
str + str	str	Concatenate two strings
str * int	str	Replicate the string
str in str	bool	Check if a string is present in another sting
str[int]	str	Read a character at specified index position
str[int:int]	str	Extract sub-string using indices

```
DNA1 = "ATTCG"
DNA2 = "GGATC"

# Joining strings
DNA3 = DNA1 + DNA2
```

Separator

List of stings

```
DNA3 = "".join([DNA1,DNA2]) <-------> "-".join([DNA1,DNA2])

tandem_repeat = DNA1 * 5
print(tandem_repeat + " has length: " + str(len(tandem_repeat)))
```

```
ATTCGATTCGATTCGATTCGATTCG has length: 25
```

```
# Searching in strings
motif1 = "TCGAT"
motif2 = "TCCT"
print(motif1 in tandem_repeat) # TRUE
print(motif2 in tandem_repeat) # FALSE
```

Zero based !

String indexing and slicing

- You can access a specific position of the string using index
- Python string indexing starts at "0"

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
D	N	A		b	a	s	e	s	:		A	T	G	C
-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
dna = "DNA bases: ATGC"

dna # the whole string
dna[0] # the first character
dna[7] # the eighth character
dna[-1] # the last character
dna[0:3] # the first three characters
dna[-4:] # the final four characters
dna[::4] # every fourth character from the beginning
dna[::-4] # every fourth character from the end
```

Output:

```
DNA bases: ATGC
D
e (not s)
C
DNA
ATGC
DbsT
C sA
```

String indexing and slicing

Zero based
Right end exclusive

- You can access a specific position of the string using index
- Python string indexing starts at "0“, and excludes right end

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
D	N	A		b	a	s	e	s	:		A	T	G	C
-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
dna = "DNA bases: ATGC"

dna # the whole string                                    DNA bases: ATGC
dna[0] # the first character                              D
dna[7] # the eighth character                             e (not s)
dna[-1] # the last character                              C
dna[0:3] # the first three characters                     DNA
dna[-4:] # the final four characters                      ATGC
dna[::4] # every fourth character from the beginning      DbsT
dna[::-4] # every fourth character from the end           C sA
```

Output:

Methods for strings

Result	Method	Meaning
str	str.upper()	Return the string in upper case
str	str.lower()	Return the string in lower case
str	str.strip(str)	Remove strings from the sides
str	str.lstrip(str)	Remove strings from the left
str	str.rstrip(str)	Remove strings from the right
str	str.replace(str, str)	Replace substrings
bool	str.startswith(str)	Check if the string starts with another
bool	str.endswith(str)	Check if the string ends with another
int	str.find(str)	Return the first position of a substring starting from the left
int	str.rfind(str)	Return the position of a substring starting from the right
int	str.count(str)	Count the number of occurrences of a substring

```
dna = "GAGTCCGTACCG"
```

Output:

```
dna.lower()                "gagtccgtaccg"
dna.strip("G")             "AGTCCGTACC"
dna.replace("G","*")       "*A*TCC*TACC*"
dna.find("TCC")            3
dna.count("G")             4
```

(also remember string operators mentioned previously!)

Important: since strings are immutable, every operation on a string actually produces a new `str` object. Thus, `dna.lower()` returns a new string, not changing the original string `dna`:

```
new_dna = dna.lower()
```

More information on string methods at: https://docs.python.org/3/library/stdtypes.html#string-methods

Methods for strings

Result	Method	Meaning
str	str.upper()	Return the string in upper case
str	str.lower()	Return the string in lower case
str	str.strip(str)	Remove strings from the sides
str	str.lstrip(str)	Remove strings from the left
str	str.rstrip(str)	Remove strings from the right
str	str.replace(str, str)	Replace substrings
bool	str.startswith(str)	Check if the string starts with another
bool	str.endswith(str)	Check if the string ends with another
int	str.find(str)	Return the first position of a substring starting from the left
int	str.rfind(str)	Return the position of a substring starting from the right
int	str.count(str)	Count the number of occurrences of a substring

```
dna = "GAGTCCGTACCG"
```

Output:

```
dna.lower()                "gagtccgtaccg"
dna.strip("G")             "AGTCCGTACC"
dna.replace("G","*")       "*A*TCC*TACC*"
dna.find("TCC")            3
dna.count("G")             4
```

(also remember string operators mentioned previously!)

Important: since strings are immutable, every operation on a string actually produces a new `str` object.
Thus, `dna.lower()` returns a new string, not changing the original string `dna`:

```
new_dna = dna.lower()
```

More information on string methods at: https://docs.python.org/3/library/stdtypes.html#string-methods

Methods for strings

Result	Method	Meaning
str	str.upper()	Return the string in upper case
str	str.lower()	Return the string in lower case
str	str.strip(str)	Remove strings from the sides
str	str.lstrip(str)	Remove strings from the left
str	str.rstrip(str)	Remove strings from the right
str	str.replace(str, str)	Replace substrings
bool	str.startswith(str)	Check if the string starts with another
bool	str.endswith(str)	Check if the string ends with another
int	str.find(str)	Return the first position of a substring starting from the left
int	str.rfind(str)	Return the position of a substring starting from the right
int	str.count(str)	Count the number of occurrences of a substring

```
dna = "GAGTCCGTACCG"
```

Output:

```
dna.lower()              "gagtccgtaccg"
dna.strip("G")           "AGTCCGTACC"
dna.replace("G","*")     "*A*TCC*TACC*"
dna.find("TCC")          3
dna.count("G")           4
```

(also remember string operators mentioned previously!)

Important: since strings are immutable, every operation on a string actually produces a new `str` object. Thus, `dna.lower()` returns a new string, not changing the original string `dna`:

```
new_dna = dna.lower()
```

More information on string methods at: https://docs.python.org/3/library/stdtypes.html#string-methods

Many ways of inserting variables into strings

```
age1 = 16
age2 = 16.5734
```

Concatenate strings using **"+"** *operator*

```
"I am " + str(age1) + " years old and my friend is " + str(age2) + " years old!"
```

"f-string" *: new notation introduced in Python 3*

```
f"I am {age1} years old and my friend is {age2} years old!"
```

Old styles with **"%"** *or* **".format"**

```
"I am %d years old and my friend is %f years old!" % (age1, age2)
"I am {0} years old and my friend is {1} years old!".format(age1, age2)
```

Many ways of inserting variables into strings

```
age1 = 16
age2 = 16.5734
```

Concatenate strings using **"+"** *operator*

```
"I am " + str(age1) + " years old and my friend is " + str(age2) + " years old!"
```

Use this one!

"f-string" *: new notation introduced in Python 3*

```
f"I am {age1} years old and my friend is {age2} years old!"
```

Old styles with **"%"** *or* **".format"**

```
"I am %d years old and my friend is %f years old!" % (age1, age2)
"I am {0} years old and my friend is {1} years old!".format(age1, age2)
```

f-string = formatted string

```
age1 = 16
age2 = 16.5734
```

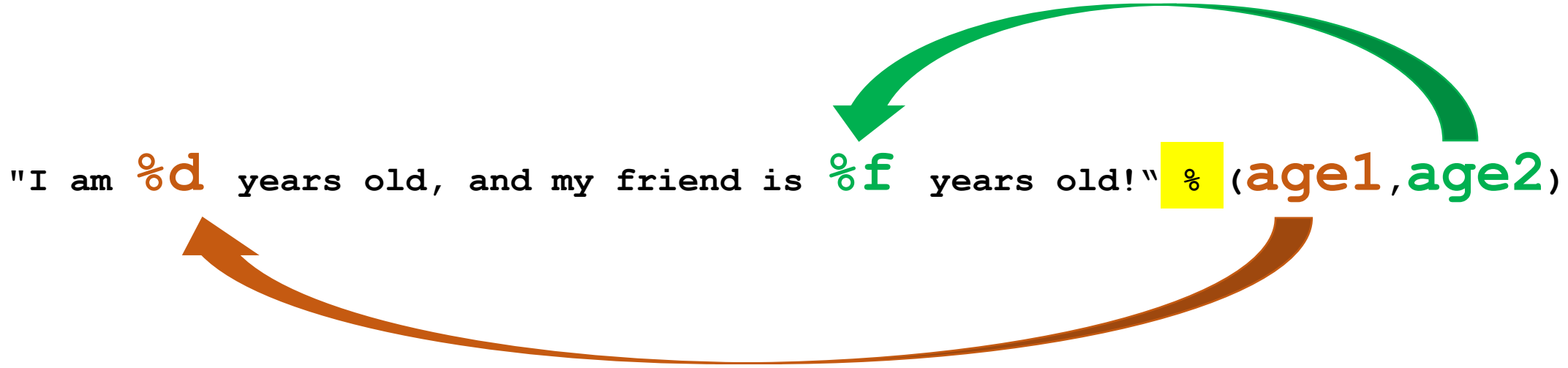
decimal ***integer*** (as opposed to **b**inary or he**x**adecimal)

```
print(f"I am {age1:d} years old, and my friend is {age2:.2f} years old!")
```

Number of digits after the dot in **f**loat

*I am **16** years old, and my friend is **16.57** years old!*

Inserting variables into string: old notations




```
"I am {} years old, and my friend is {} years old!".format(age1, age2)
"I am {0} years old, and my friend is {1} years old!".format(age1, age2)
"I am {0:d} years old, and my friend is {1:.2f} years old!".format(age1, age2)
```

Lecture plan (learning outcomes)

At end of this lecture, you will be able to:

- Operate simple Python functions for command line input/output
- Understand Python variables and built-in data types
- Describe operations and functions for Boolean and numeric data types
- Understand strings indexing and perform operations with strings
- Control `print()` function

`print()` : controlling the end of line

By default, `print()` adds a new line at the end, which can be changed …

Code

```
print("DNA bases: ATGC")
print("RNA bases: AUGC")
```

Output

```
DNA bases: ATGC
RNA bases: AUGC
```

```
print("DNA bases: ATGC", end = "")
print("RNA bases: AUGC")
```

```
DNA bases: ATGCRNA bases: AUGC
```

```
print("DNA bases: ATGC", end = " ")
print("RNA bases: AUGC")
```

```
DNA bases: ATGC RNA bases: AUGC
```

Printing multiple variables

Just use comma inside the print() command !

or use any method of inserting variables into strings discussed previously :)

```
age1 = input("What is your age?:")  # 16
age2 = input("What is your friend's age?:")  # 16.5734
print("I am", age1, "years old, and my friend is", age2, "years old!")
```

I am 16 years old, and my friend is 16.5734 years old!

Lecture plan (learning outcomes)

At end of this lecture, you will be able to:

- Operate simple Python functions for command line input/output
- Understand Python variables and built-in data types
- Describe operations and functions for Boolean and numeric data types
- Understand strings indexing and perform operations with strings
- Control `print()` function

Questions