Practical Computing for Scientists

Armin Sobhani

CSCI 2000U

UOIT – Fall 2015

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Python

Basics

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Numbers

Numbers

14

32-bit integer

(on most machines)

Numbers

14	32-bit integer (on most machines)
14.0	64-bit float (ditto)

Numbers

14	32-bit integer (on most machines)
14.0	64-bit float (ditto)
1+4j	complex number (two 64-bit floats)

Numbers

14	32-bit integer (on most machines)
14.0	64-bit float (ditto)
1+4j	complex number (two 64-bit floats)
x.real, x.imag	real and imaginary parts of complex number

Arithmetic

Arithmetic

Addition	+	35 + 22	57

Arithmetic

Addition	+	`35 + 22`	`57`
		`'Py' + 'thon'`	`'Python'`

Arithmetic

Addition	+	`35 + 22`	`57`
		`'Py' + 'thon'`	`'Python'`
Subtraction	–	`35 - 22`	`13`

Arithmetic

Addition	+	35 + 22	57
		'Py' + 'thon'	'Python'
Subtraction	–	35 - 22	13
Multiplication	*	3 * 2	6

Arithmetic

Addition	+	35 + 22	57
		'Py' + 'thon'	'Python'
Subtraction	–	35 - 22	13
Multiplication	*	3 * 2	6
		'Py' * 2	'PyPy'

Arithmetic

Addition	+	`35 + 22`	`57`
		`'Py' + 'thon'`	`'Python'`
Subtraction	–	`35 - 22`	`13`
Multiplication	*	`3 * 2`	`6`
		`'Py' * 2`	`'PyPy'`
Division	/	`3.0 / 2`	`1.5`

Arithmetic

Addition	+	35 + 22	57
		'Py' + 'thon'	'Python'
Subtraction	–	35 - 22	13
Multiplication	*	3 * 2	6
		'Py' * 2	'PyPy'
Division	/	3.0 / 2	1.5
		3 / 2	2.x: 1 3.x: 1.5

Arithmetic

Addition	+	35 + 22	57
		'Py' + 'thon'	'Python'
Subtraction	–	35 - 22	13
Multiplication	*	3 * 2	6
		'Py' * 2	'PyPy'
Division	/	3.0 / 2	1.5
		3 / 2	2.x: 1 3.x: 1.5
Exponentiation	**	2 ** 0.5	1.41421356...

Arithmetic

Addition	+	35 + 22	57
		'Py' + 'thon'	'Python'
Subtraction	-	35 - 22	13
Multiplication	*	3 * 2	6
		'Py' * 2	'PyPy'
Division	/	3.0 / 2	1.5
		3 / 2	2.x: 1 3.x: 1.5
Exponentiation	**	2 ** 0.5	1.41421356...
Remainder	%	13 % 5	3

Prefer *in-place* forms of binary operators

Prefer *in-place* forms of binary operators

```
>>> years = 500
>>>
```

Prefer *in-place* forms of binary operators

```
>>> years = 500
>>> years += 1
>>>
```

Prefer *in-place* forms of binary operators

```
>>> years = 500
>>> years += 1
>>>
```

The same as `years = years + 1`

Prefer *in-place* forms of binary operators

```
>>> years = 500
>>> years += 1
>>> print(years)
501
>>>
```

Prefer *in-place* forms of binary operators

```
>>> years = 500
>>> years += 1
>>> print(years)
501
>>> years %= 10
>>>
```

Prefer *in-place* forms of binary operators

```
>>> years = 500
>>> years += 1
>>> print(years)
501
>>> years %= 10
>>>
```

The same as `years = years % 10`

Prefer *in-place* forms of binary operators

```
>>> years = 500
>>> years += 1
>>> print(years)
501
>>> years %= 10
>>> print(years)
1
>>>
```

Comparisons

Comparisons

3 < 5

True

Comparisons

3 < 5	True
3 != 5	True

Comparisons

3 < 5	True
3 != 5	True
3 == 5	False

Comparisons

3 < 5	True
3 != 5	True
3 == 5	False

Single = is assignment

Double == is equality

Comparisons

3 < 5	True
3 != 5	True
3 == 5	False
3 >= 5	False

Comparisons

3 < 5	True
3 != 5	True
3 == 5	False
3 >= 5	False
1 < 3 < 5	True

Comparisons

3 < 5	True
3 != 5	True
3 == 5	False
3 >= 5	False
1 < 3 < 5	True
1 < 5 > 3	True

But please don't do this

Comparisons

`3 < 5`	True
`3 != 5`	True
`3 == 5`	False
`3 >= 5`	False
`1 < 3 < 5`	True
`1 < 5 > 3`	True
`3+2j < 5`	*error*

Python

Control Flow

by Greg Wilson

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

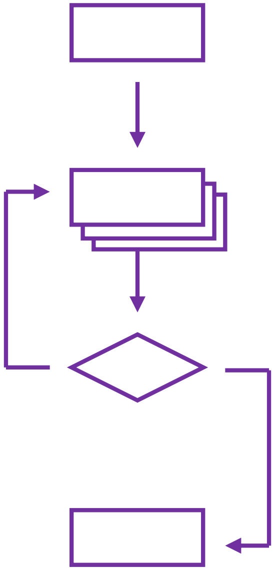
Real power of programs comes from:

Real power of programs comes from:

repetition

Real power of programs comes from:

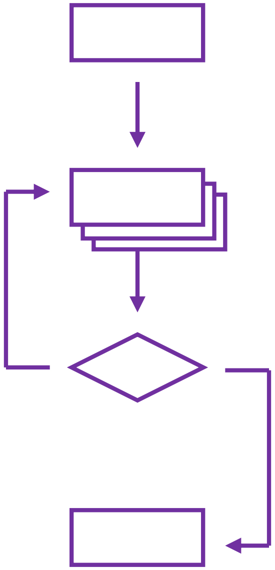
repetition



Real power of programs comes from:

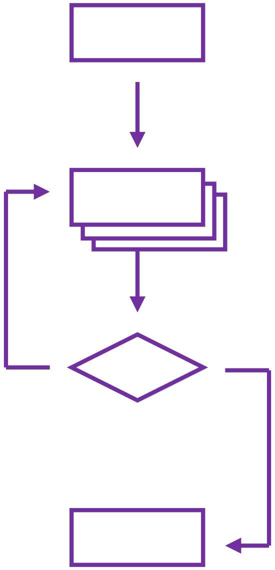
repetition

selection

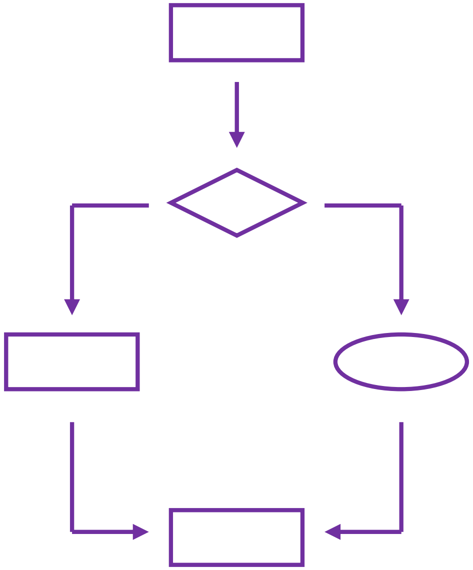


Real power of programs comes from:

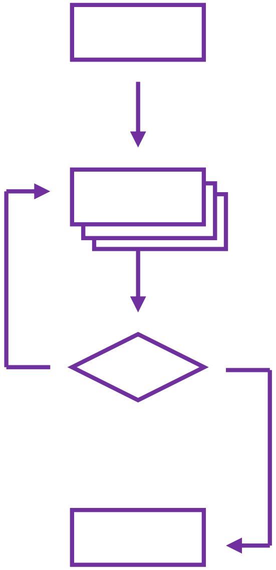
repetition



selection



Repetition



Simplest form of repetition is *while loop*

Simplest form of repetition is *while loop*

```
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```

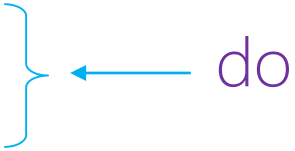
Simplest form of repetition is *while loop*

```
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```

test

Simplest form of repetition is *while loop*

```
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```



do

Simplest form of repetition is *while loop*

```
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```

do

```
3
```

Simplest form of repetition is *while loop*

```
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```

test again

```
3
```

Simplest form of repetition is *while loop*

```
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```

```
3
2
```

Simplest form of repetition is *while loop*

```python
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```

```
3
2
1
```

While loop may execute zero times

While loop may execute zero times

```
print('before')
num_moons = -3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
print('after')
```

While loop may execute zero times

```
print('before')
num_moons = -3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
print('after')
```

not true when first tested…

While loop may execute zero times

```
print('before')
num_moons = -3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
print('after')
```

...so this is never executed

While loop may execute zero times

```
print('before')
num_moons = -3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
print('after')
```

```
before
after
```

While loop may execute zero times

```python
print('before')
num_moons = -3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
print('after')
```

```
before
after
```

Important to consider this case when designing and testing code

While loop may also execute forever

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

While loop may also execute forever

```python
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
```

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
3
```

While loop may also execute forever

```python
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
3
3
```

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
3
3
3
```

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
3
3
3
⋮
```

While loop may also execute forever

```python
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
3
3
3
⋮
```

Nothing in here changes the loop control condition

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
3
3
3
⋮
```

Usually not the desired behavior…

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
before
3
3
3
⋮
```

Usually not the desired behavior…

…but there *are* cases where it's useful

Why indentation?

Why indentation?

Studies show that's what people actually pay attention to

Why indentation?

Studies show that's what people actually pay attention to

- Every textbook on C or Java has examples where indentation and braces don't match

Why indentation?

Studies show that's what people actually pay attention to

- Every textbook on C or Java has examples where indentation and braces don't match

Doesn't matter how much you use, but whole block must be consistent

Why indentation?

Studies show that's what people actually pay attention to

- Every textbook on C or Java has examples where indentation and braces don't match

Doesn't matter how much you use, but whole block must be consistent

Python Style Guide (PEP 8) recommends 4 spaces

Why indentation?

Studies show that's what people actually pay attention to

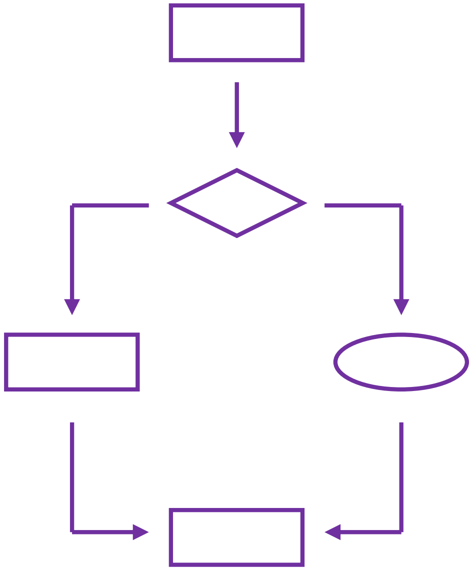
- Every textbook on C or Java has examples where indentation and braces don't match

Doesn't matter how much you use, but whole block must be consistent

Python Style Guide (PEP 8) recommends 4 spaces

And no tab characters

Selection



Use `if`, `elif`, and `else` to make choices

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

Use **`if`**, **`elif`**, and **`else`** to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

← not true when first tested…

Use `if`, `elif`, and `else` to make choices

```python
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

← …so this is *not* executed

Use **if**, **elif**, and **else** to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

this isn't true either...

Use **if**, **elif**, and **else** to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')  ←  ...so this isn't executed
else:
    print('greater')
```

Use **if**, **elif**, and **else** to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

← nothing else has executed...

Use **if**, **elif**, and **else** to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

← ...so this *is* executed

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

```
greater
```

Use **if**, **elif**, and **else** to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
greater
```

Always start with **if**

Use **`if`**, **`elif`**, and **`else`** to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
greater
```

Always start with **`if`**

Can have any number of **`elif`** clauses (including none)

Use **if**, **elif**, and **else** to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

```
greater
```

Always start with **if**

Can have any number of **elif** clauses (including none)

And the **else** clause is optional

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
greater
```

Always start with `if`

Can have any number of `elif` clauses (including none)

And the `else` clause is optional

Always tested in order

Blocks may contain blocks

Blocks may contain blocks

```
num = 0
while num <= 10:
    if (num % 2) == 1:
        print(num)
    num += 1
```

Blocks may contain blocks

```
num = 0
while num <= 10:
    if (num % 2) == 1:
        print(num)
    num += 1
```

Count from 0 to 10

Blocks may contain blocks

```
num = 0
while num <= 10:
    if (num % 2) == 1:
        print(num)
    num += 1
```

Print odd numbers

Blocks may contain blocks

```
num = 0
while num <= 10:
    if (num % 2) == 1:
        print(num)
    num += 1
```

```
1
3
5
7
9
```

A better way to do it

A better way to do it

```
num = 1
while num <= 10:
    print(num)
    num += 2
```

A better way to do it

```
num = 1
while num <= 10:
    print(num)
    num += 2
```

```
1
3
5
7
9
```

Writing a simple program that *works*,

Writing a simple program that *works*,
then tweaking it to make it *more efficient*,

Writing a simple program that *works*,
then tweaking it to make it *more efficient*,
is a common *pattern* in programming.

Writing a simple program that *works*,
then tweaking it to make it *more efficient*,
is a common *pattern* in programming.

Another is to write programs *top-down*,

Writing a simple program that *works*,
then tweaking it to make it *more efficient*,
is a common *pattern* in programming.

Another is to write programs *top-down*,
solving one problem at a time.

Print primes less than 1000

Print primes less than 1000

```
num = 2
while num <= 1000:
    ...figure out if num is prime...
    if is_prime:
        print(num)
    num += 1
```

Print primes less than 1000

```
num = 2
while num <= 1000:
    ...figure out if num is prime...
    if is_prime:
        print(num)
    num += 1
```

Cannot be evenly divided

by any other integer

Print primes less than 1000

```
num = 2
while num <= 1000:
    ...figure out if num is prime...
    if is_prime:
        print(num)
    num += 1
```

```
is_prime = True
trial = 2
while trial < num:
    if ...num divisible by trial...:
        is_prime = False
    trial += 1
```

Print primes less than 1000

```
num = 2
while num <= 1000:
    ...figure out if num is prime...
    if is_prime:
        print(num)
    num += 1
```

Remainder is zero

```
is_prime = True
trial = 2
while trial < num:
    if ...num divisible by trial...:
        is_prime = False
    trial += 1
```

Print primes less than 1000

```
num = 2
while num <= 1000:
    ...figure out if num is prime...
    if is_prime:
        print(num)
    num += 1
```

```
(num % trial) == 0
```

```
is_prime = True
trial = 2
while trial < num:
    if ...num divisible by trial...:
        is_prime = False
    trial += 1
```

Print primes less than 1000

```
num = 2
while num <= 1000:
    is_prime = True
    trial = 2
    while trial < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

Print primes less than 1000 (more efficient version)

Print primes less than 1000 (more efficient version)

```
num = 2
while num <= 1000:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

Print primes less than 1000 (more efficient version)

```
num = 2
while num <= 1000:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

testing whether n is multiple of any integer between 2 and $\sqrt{n}$

Any code that hasn't been tested is probably wrong

Any code that hasn't been tested is probably wrong

```
num = 2
while num <= 10:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

Any code that hasn't been tested is probably wrong

```python
num = 2
while num <= 10:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

```
2
3
4
5
7
9
```

Any code that hasn't been tested is probably wrong

```
num = 2
while num <= 10:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

2

3

$\boxed{4}$

5

7

$\boxed{9}$

Any code that hasn't been tested is probably wrong

```
num = 2
while num <= 10:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

```
2
3
4
5
7
9
```

Where's the bug?

Python

Lists

UNIVERSITY OF ONTARIO
INSTITUTE OF TECHNOLOGY

Loops let us do things many times

Loops let us do things many times

Collections let us store many values together

Loops let us do things many times

Collections let us store many values together

Most popular collection is a *list*

Create using [value, value, …]

Create using [value, value, ...]

Get/set values using var[index]

Create using [value, value, ...]

Get/set values using var[index]

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(gases)
['He', 'Ne', 'Ar', 'Kr']
```

Create using [value, value, ...]

Get/set values using var[index]

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(gases)
['He', 'Ne', 'Ar', 'Kr']

print(gases[1])
Ne
```

Index from 0, not 1

Index from 0, not 1

Reasons made sense for C in 1970…

Index from 0, not 1

Reasons made sense for C in 1970…

It's an error to try to access out of range

Index from 0, not 1

Reasons made sense for C in 1970...

It's an error to try to access out of range

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(gases[4])
IndexError: list index out of range
```

Use len(list) to get length of list

Use len(list) to get length of list

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(len(gases))
4
```

Use len(list) to get length of list

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(len(gases))
4
```

Returns 0 for the *empty* list

```
etheric = []
print(len(etheric))
0
```

Some negative indices work

Some negative indices work

values[-1] is last element, values[-2] next-to-last, …

Some negative indices work

values[-1] is last element, values[-2] next-to-last, ...

```
gases = ['He', 'Ne', 'Ar', 'Kr']
```

Some negative indices work

values[-1] is last element, values[-2] next-to-last, ...

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(gases[-1], gases[-4])
Kr He
```

Some negative indices work

values[-1] is last element, values[-2] next-to-last, ...

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(gases[-1], gases[-4])
Kr He
```

values[-1] is much nicer than values[len(values)-1]

Some negative indices work

values[-1] is last element, values[-2] next-to-last, ...

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(gases[-1], gases[-4])
Kr He
```

values[-1] is much ~~nicer~~ than values[len(values)-1]

less error prone

Mutable : can change it after it is created

Mutable : can change it after it is created

```
gases = ['He', 'Ne', 'Ar', 'K']  # last entry misspelled
```

Mutable : can change it after it is created

```
gases = ['He', 'Ne', 'Ar', 'K']  # last entry misspelled
gases[3] = 'Kr'
```

Mutable : can change it after it is created

```
gases = ['He', 'Ne', 'Ar', 'K']  # last entry misspelled
gases[3] = 'Kr'
print(gases)
['He', 'Ne', 'Ar', 'Kr']
```

Mutable : can change it after it is created

```
gases = ['He', 'Ne', 'Ar', 'K']  # last entry misspelled
gases[3] = 'Kr'
print(gases)
['He', 'Ne', 'Ar', 'Kr']
```

Location must exist before assignment

Mutable : can change it after it is created

```
gases = ['He', 'Ne', 'Ar', 'K']  # last entry misspelled
gases[3] = 'Kr'
print(gases)
['He', 'Ne', 'Ar', 'Kr']
```

Location must exist before assignment

```
gases = ['He', 'Ne', 'Ar', 'Kr']
```

Mutable : can change it after it is created

```
gases = ['He', 'Ne', 'Ar', 'K']  # last entry misspelled
gases[3] = 'Kr'
print(gases)
['He', 'Ne', 'Ar', 'Kr']
```

Location must exist before assignment

```
gases = ['He', 'Ne', 'Ar', 'Kr']
gases[4] = 'Xe'
IndexError: list assignment index out of range
```

Heterogeneous : can store values of many kinds

Heterogeneous : can store values of many kinds

```
helium = ['He', 2]
neon = ['Ne', 8]
```

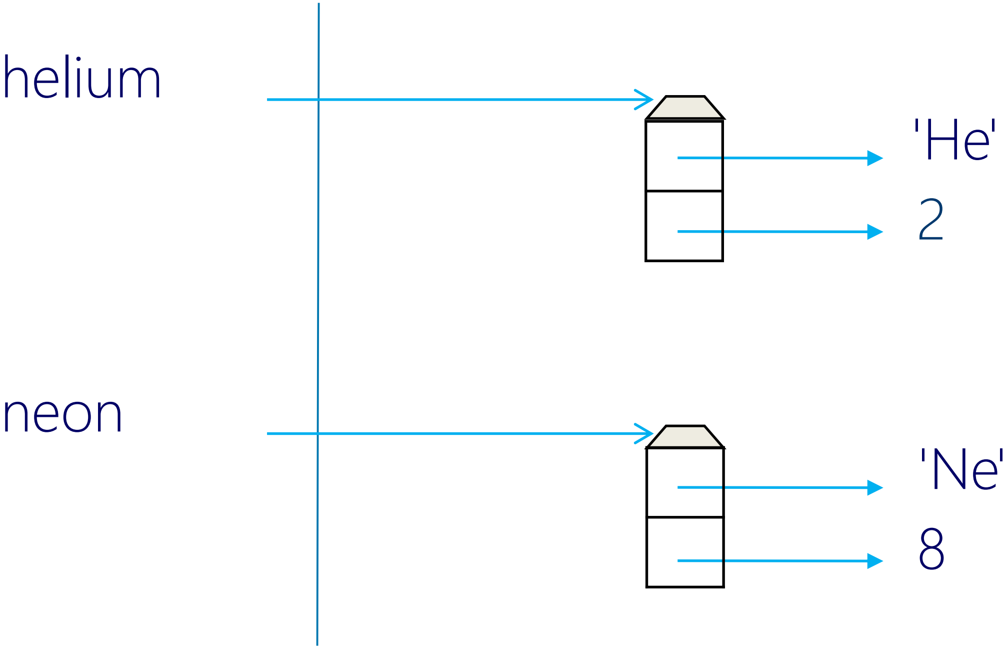
Heterogeneous : can store values of many kinds

```
helium = ['He', 2]
neon = ['Ne', 8]
```

[string, int]

Heterogeneous : can store values of many kinds

```
helium = ['He', 2]
neon = ['Ne', 8]
```

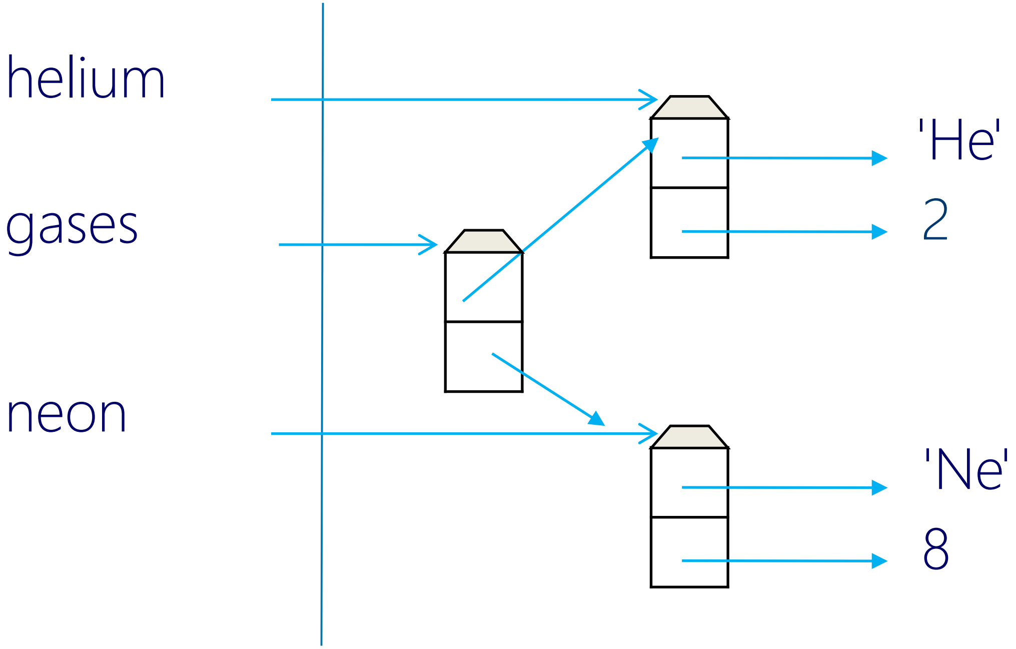


Heterogeneous : can store values of many kinds

```
helium = ['He', 2]
neon = ['Ne', 8]
gases = [helium, neon]
```

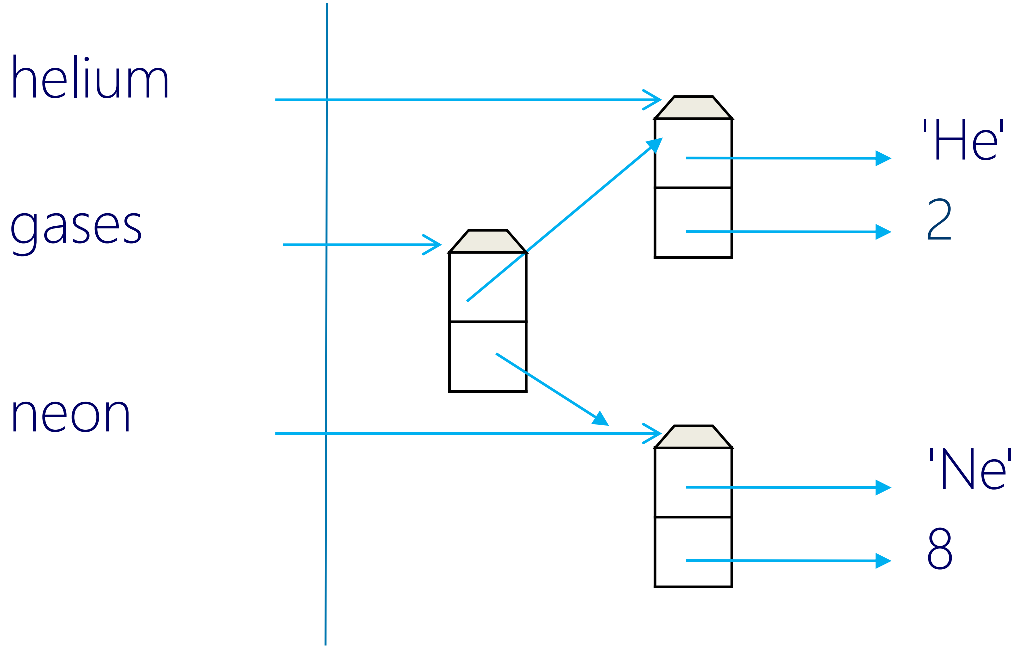
Heterogeneous : can store values of many kinds

```
helium = ['He', 2]
neon = ['Ne', 8]
gases = [helium, neon]
```



Heterogeneous : can store values of many kinds

```
helium = ['He', 2]
neon = ['Ne', 8]
gases = [helium, neon]
```



very powerful

feature

Loop over elements to "do all"

Loop over elements to "do all"

Use while to step through all possible indices

Loop over elements to "do all"

Use while to step through all possible indices

```
gases = ['He', 'Ne', 'Ar', 'Kr']
i = 0
while i < len(gases):
    print(gases[i])
    i += 1
```

Loop over elements to "do all"

Use while to step through all possible indices

```
gases = ['He', 'Ne', 'Ar', 'Kr']
i = 0
while i < len(gases):
    print(gases[i])
    i += 1
```

First legal index

Loop over elements to "do all"

Use while to step through all possible indices

```
gases = ['He', 'Ne', 'Ar', 'Kr']
i = 0
while i < len(gases):
    print(gases[i])
    i += 1
```

Next index

Loop over elements to "do all"

Use while to step through all possible indices

```
gases = ['He', 'Ne', 'Ar', 'Kr']
i = 0
while i < len(gases):
    print(gases[i])
    i += 1
```

Defines set of legal indices

Loop over elements to "do all"

Use while to step through all possible indices

```
gases = ['He', 'Ne', 'Ar', 'Kr']
i = 0
while i < len(gases):
    print(gases[i])
    i += 1
```

```
He
Ne
Ar
Kr
```

Loop over elements to "do all"

Use while to step through all possible indices

```
gases = ['He', 'Ne', 'Ar', 'Kr']
i = 0
while i < len(gases):
    print(gases[i])
    i += 1
```

```
He
Ne
Ar
Kr
```

Tedious to type in over and over again

Loop over elements to "do all"

Use while to step through all possible indices

```
gases = ['He', 'Ne', 'Ar', 'Kr']
i = 0
while i < len(gases):
    print(gases[i])
    i += 1
```

```
He
Ne
Ar
Kr
```

Tedious to type in over and over again

And it's easy to forget the "+= 1" at the end

Use a for loop to access each value in turn

Use a for loop to access each value in turn

```
gases = ['He', 'Ne', 'Ar', 'Kr']
for gas in gases:
    print(gas)
He
Ne
Ar
Kr
```

Use a for loop to access each value in turn

```
gases = ['He', 'Ne', 'Ar', 'Kr']
for gas in gases:
    print(gas)
He
Ne
Ar
Kr
```

Loop variable assigned each value in turn

Use a for loop to access each value in turn

```
gases = ['He', 'Ne', 'Ar', 'Kr']
for gas in gases:
    print(gas)
He
Ne
Ar
Kr
```

Loop variable assigned each value in turn

Not each index

Use a for loop to access each value in turn

```
gases = ['He', 'Ne', 'Ar', 'Kr']
for gas in gases:
    print(gas)
He
Ne
Ar
Kr
```

Loop variable assigned each value in turn

Not each index

Because that's the most common case

Can delete entries entirely (shortens the list)

Can delete entries entirely (shortens the list)

```
gases = ['He', 'Ne', 'Ar', 'Kr']
```

Can delete entries entirely (shortens the list)

```
gases = ['He', 'Ne', 'Ar', 'Kr']
del gases[0]
```

Can delete entries entirely (shortens the list)

```
gases = ['He', 'Ne', 'Ar', 'Kr']
del gases[0]
print(gases)
['Ne', 'Ar', 'Kr']
```

Can delete entries entirely (shortens the list)

```
gases = ['He', 'Ne', 'Ar', 'Kr']
del gases[0]
print(gases)
['Ne', 'Ar', 'Kr']
del gases[2]
```

Can delete entries entirely (shortens the list)

```
gases = ['He', 'Ne', 'Ar', 'Kr']
del gases[0]
print(gases)
['Ne', 'Ar', 'Kr']
del gases[2]
print(gases)
['Ne', 'Ar']
```

Can delete entries entirely (shortens the list)

```
gases = ['He', 'Ne', 'Ar', 'Kr']
del gases[0]
print(gases)
['Ne', 'Ar', 'Kr']
del gases[2]
print(gases)
['Ne', 'Ar']
```

Yes, deleting an index that doesn't exist is an error

Appending values to a list lengthens it

Appending values to a list lengthens it

```
gases = []
```

Appending values to a list lengthens it

```
gases = []
gases.append('He')
```

Appending values to a list lengthens it

```
gases = []
gases.append('He')
gases.append('Ne')
```

Appending values to a list lengthens it

```
gases = []
gases.append('He')
gases.append('Ne')
gases.append('Ar')
```

Appending values to a list lengthens it

```
gases = []
gases.append('He')
gases.append('Ne')
gases.append('Ar')
print(gases)
['He', 'Ne', 'Ar']
```

Appending values to a list lengthens it

```
gases = []
gases.append('He')
gases.append('Ne')
gases.append('Ar')
print(gases)
['He', 'Ne', 'Ar']
```

Most operations on lists are methods

Appending values to a list lengthens it

```
gases = []
gases.append('He')
gases.append('Ne')
gases.append('Ar')
print(gases)
['He', 'Ne', 'Ar']
```

Most operations on lists are methods

A function that belongs to (and usually operates on) specific data

Appending values to a list lengthens it

```
gases = []
gases.append('He')
gases.append('Ne')
gases.append('Ar')
print(gases)
['He', 'Ne', 'Ar']
```

Most operations on lists are methods

A function that belongs to (and usually operates on) specific data

thing . method (args)

Some useful list methods

Some useful list methods

```python
gases = ['He', 'He', 'Ar', 'Kr'] # 'He' is duplicated
```

Some useful list methods

```
gases = ['He', 'He', 'Ar', 'Kr'] # 'He' is duplicated
print(gases.count('He'))
2
```

Some useful list methods

```
gases = ['He', 'He', 'Ar', 'Kr'] # 'He' is duplicated
print(gases.count('He'))
2
print(gases.index('Ar'))
2
```

Some useful list methods

```
gases = ['He', 'He', 'Ar', 'Kr'] # 'He' is duplicated
print(gases.count('He'))
2
print(gases.index('Ar'))
2
gases.insert(1, 'Ne')
```

Some useful list methods

```
gases = ['He', 'He', 'Ar', 'Kr'] # 'He' is duplicated
print(gases.count('He'))
2
print(gases.index('Ar'))
2
gases.insert(1, 'Ne')
print(gases)
['He', 'Ne', 'He', 'Ar', 'Kr']
```

Two that are often used incorrectly

Two that are often used incorrectly

```
gases = ['He', 'Ne', 'Ar', 'Kr']
```

Two that are often used incorrectly

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(gases.sort())
None
```

Two that are often used incorrectly

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(gases.sort())
None
print(gases)
['Ar', 'He', 'Kr', 'Ne']
```

Two that are often used incorrectly

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(gases.sort())
None
print(gases)
['Ar', 'He', 'Kr', 'Ne']
print(gases.reverse())
None
```

Two that are often used incorrectly

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(gases.sort())
None
print(gases)
['Ar', 'He', 'Kr', 'Ne']
print(gases.reverse())
None
print(gases)
['Ne', 'Kr', 'He', 'Ar']
```

Two that are often used incorrectly

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(gases.sort())
None
print(gases)
['Ar', 'He', 'Kr', 'Ne']
print(gases.reverse())
None
print(gases)
['Ne', 'Kr', 'He', 'Ar']
```

A common bug

Two that are often used incorrectly

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(gases.sort())
None
print(gases)
['Ar', 'He', 'Kr', 'Ne']
print(gases.reverse())
None
print(gases)
['Ne', 'Kr', 'He', 'Ar']
```

A common bug

gases = gases.sort() assigns None to gases

Use **`in`** to test for membership

Use **`in`** to test for membership

```
gases = ['He', 'Ne', 'Ar', 'Kr']
```

Use `in` to test for membership

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print('He' in gases)
True
```

Use **in** to test for membership

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print('He' in gases)
True
if 'Pu' in gases:
    print('But plutonium is not a gas!')
else:
    print('The universe is well ordered.')
```

Use **in** to test for membership

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print('He' in gases)
True
if 'Pu' in gases:
    print('But plutonium is not a gas!')
else:
    print('The universe is well ordered.')
The universe is well ordered.
```

Use **`range`** to construct lists of numbers

Use **range** to construct lists of numbers

```
print(range(5))
[0, 1, 2, 3, 4]
```

Use **range** to construct lists of numbers

```
print(range(5))
[0, 1, 2, 3, 4]
print(range(2, 6))
[2, 3, 4, 5]
```

Use **range** to construct lists of numbers

```
print(range(5))
[0, 1, 2, 3, 4]
print(range(2, 6))
[2, 3, 4, 5]
print(range(0, 10, 3))
[0, 3, 6, 9]
```

Use **range** to construct lists of numbers

```
print(range(5))
[0, 1, 2, 3, 4]
print(range(2, 6))
[2, 3, 4, 5]
print(range(0, 10, 3))
[0, 3, 6, 9]
print(range(10, 0))
[]
```

So **`range(len(list))`** is all indices for the list

So **range(len(list))** is all indices for the list

```
gases = ['He', 'Ne', 'Ar', 'Kr']
```

So `range(len(list))` is all indices for the list

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(len(gases))
4
```

So **range(len(list))** is all indices for the list

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(len(gases))
4
print(range(len(gases)))
[0, 1, 2, 3]
```

So **`range(len(list))`** is all indices for the list

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(len(gases))
4
print(range(len(gases)))
[0, 1, 2, 3]
for i in range(len(gases)):
    print(i, gases[i])
```

So `range(len(list))` is all indices for the list

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(len(gases))
4
print(range(len(gases)))
[0, 1, 2, 3]
for i in range(len(gases)):
    print(i, gases[i])
0 He
1 Ne
2 Ar
3 Kr
```

So **`range(len(list))`** is all indices for the list

```
gases = ['He', 'Ne', 'Ar', 'Kr']
print(len(gases))
4
print(range(len(gases)))
[0, 1, 2, 3]
for i in range(len(gases)):
    print(i, gases[i])
0 He
1 Ne
2 Ar
3 Kr
```

A very common idiom in Python