Forritunarmálið Python

Dagur 1

Kynning og grunnatriði Python

Hjalti Magnússon

27. nóvember, 2017

Course Schedule

Classes

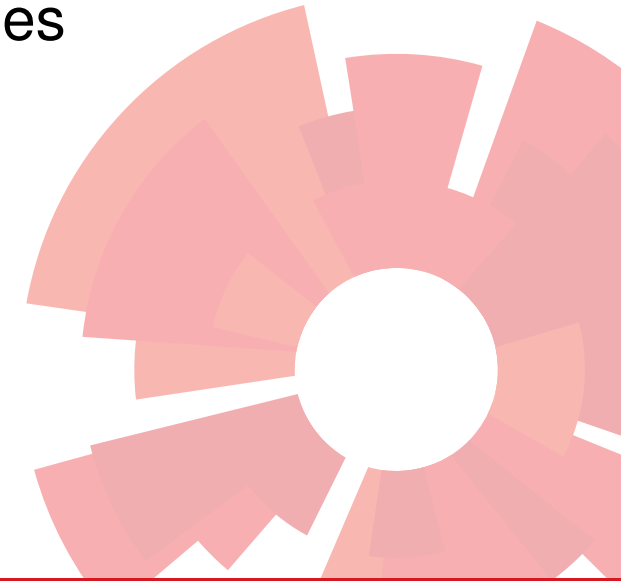
- Two lectures each day
  - In the morning and after lunch
  - Try to keep them short
- Time to work on problems between and after lectures

Evaluation

- Assignments (75%)
  - Verkefni 0 (5%)
  - Verkefni 1 (15%)
  - Verkefni 2 (30%)
  - Verkefni 3 (20%)
  - Keppni (5%)
- Final (25%)
  - **You must pass the final exam to pass the course**

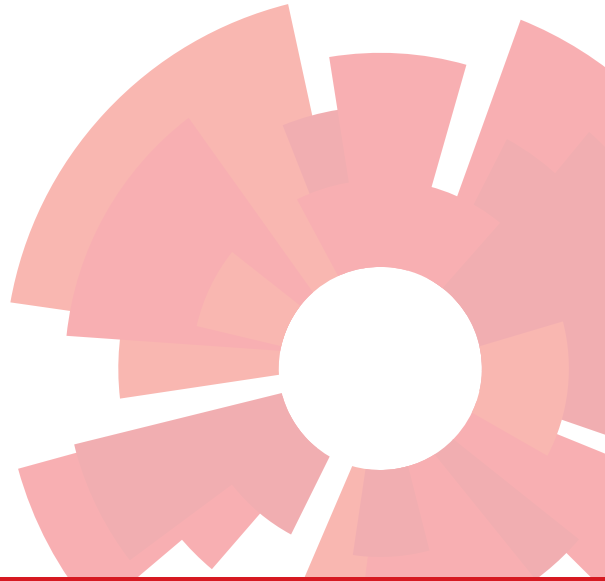
Mooshak assignments

- Only accepted solutions are considered
  - No partial scores are given for non-accepted solutions
- Accepted solutions are not automatically correct, points can be deducted for, among other things,
  - Tailoring solutions to specific test cases
  - Hard coding solutions



Book

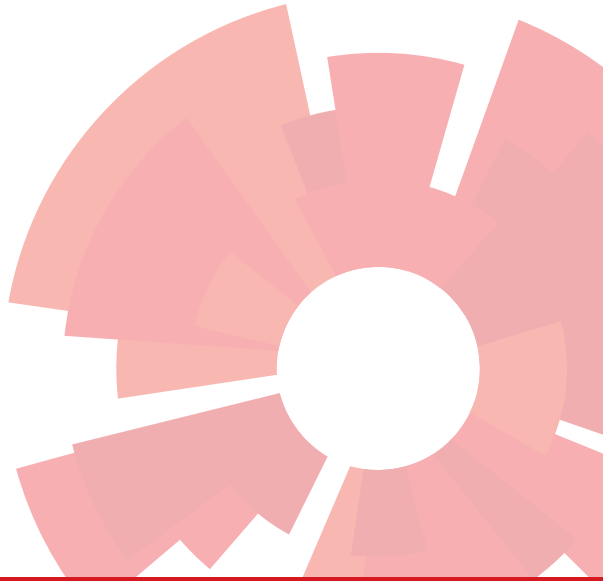
- We will loosely follow Python 101 by Michael Driscoll
- It's also good to go through The Python Tutorial



Plagiarism

Plagiarism

- All assignments in this course, except one, are individual assignments
  - The work should be yours
  - References should be cited



Similarity checker

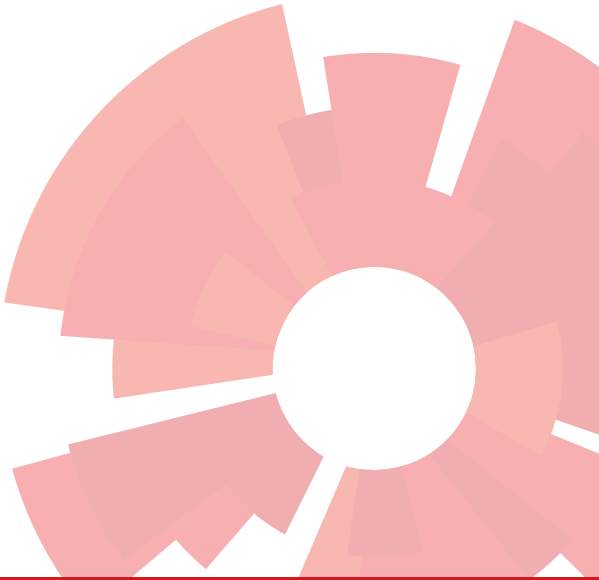
- There are solutions to some problems on the internet
  - Please do not try to look for them
- All solutions are run through a program that finds similarities with other student's solutions, solutions from previous years and to online solutions
  - Penalties range from 0 for the whole assignment, 0 for the course or expulsion, depending on how serious the violation is

How to work on the problems

- I would recommend working on the problems alone
  - It really is the best way to learn
- You are, of course, allowed to work on solving the problems together
- **However**, when writing a submission to Mooshak, you should do that alone, **from scratch**
  - That way you ensure that the solution is truly yours
- You don't have to solve all the problems

Final note

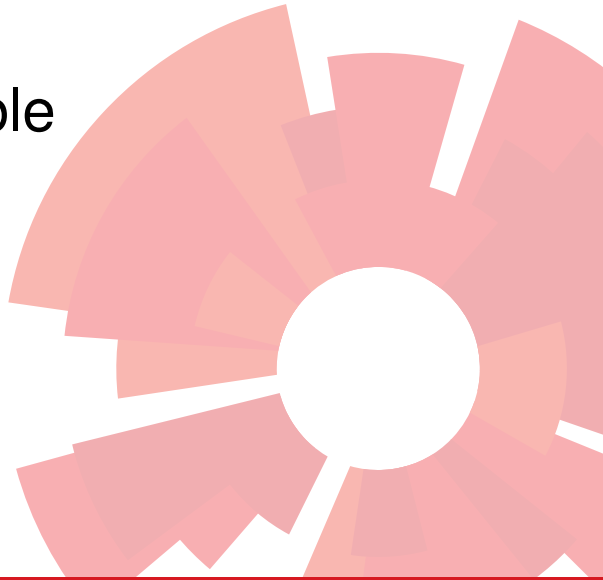
- Please do not post your solutions on an open website
  - a public Github repo
  - pastebin
  - etc



Python

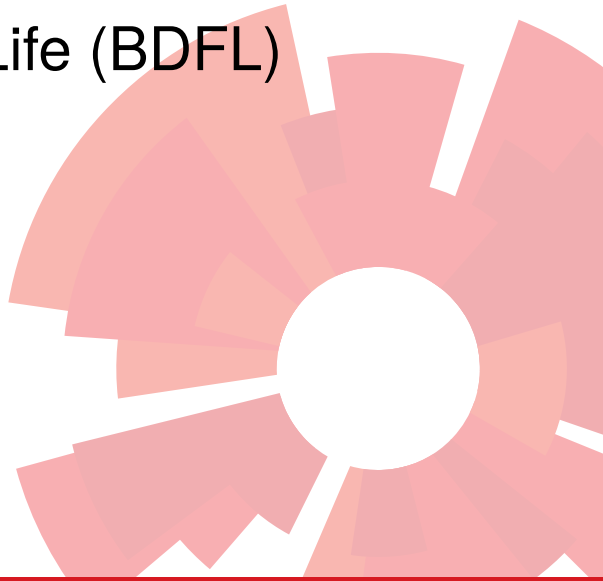
History

- Created by Guido van Rossum in 1991
- Python 2.0 released in 2000 (first “serious” release)
- Started gaining popularity around 2005
  - Adopted by Google around that time
- Python 3.0 released in 2008
  - Controversially backwards-incompatible



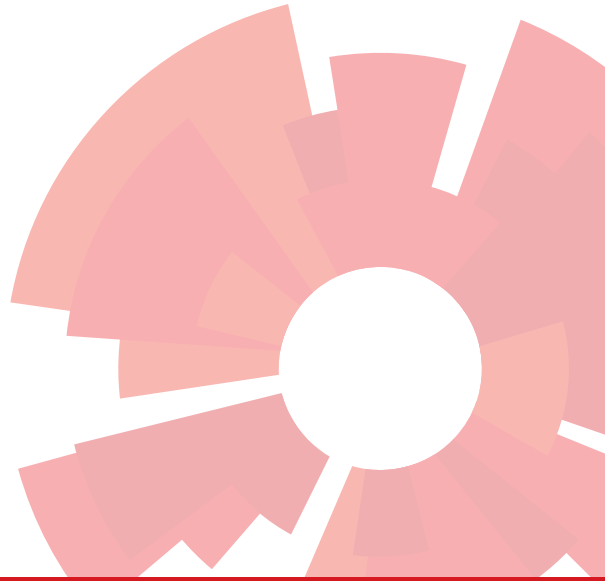
Development

- Maintained by The Python Software Foundation
- Open source effort (CPython)
- Developed by the community
  - Python Enhancement Proposals (PEP)
- Guido serves as Benevolent Dictator For Life (BDFL)



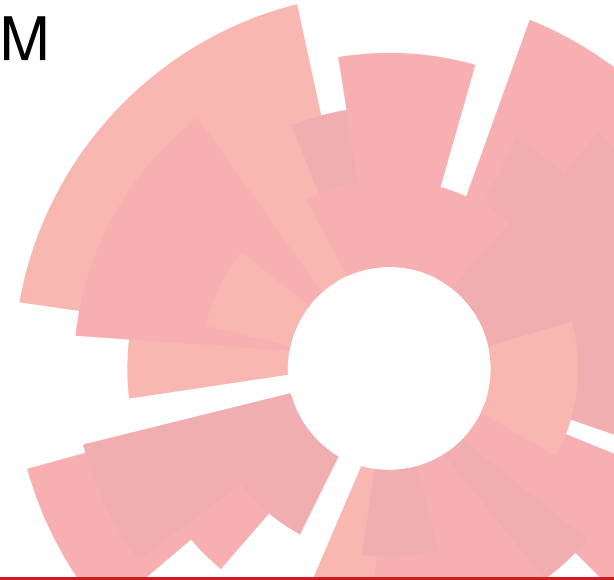
The Language

- Interpreted
  - Python bytecode
  - Interpreter written in C (CPython)
- Multi-paradigm
  - Procedural
  - Object-oriented
  - Functional



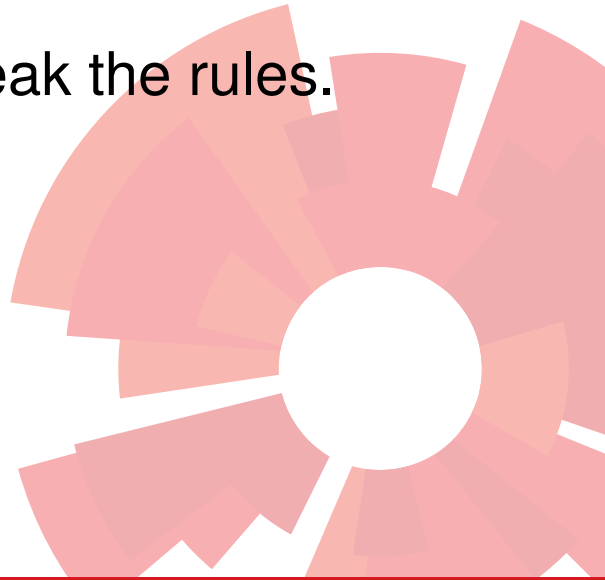
Other Things

- Python 2 vs 3
  - Things that are different between Python 2 and 3 are marked with *
- Implementations
  - CPython - The official implementation
  - IronPython - Implementation in .NET
  - Jython - Implementation using the JVM
  - PyPy - Fancy implementation (JIT)
- IDEs
  - Idle, PyCharm, Idle, Thonny, Wing



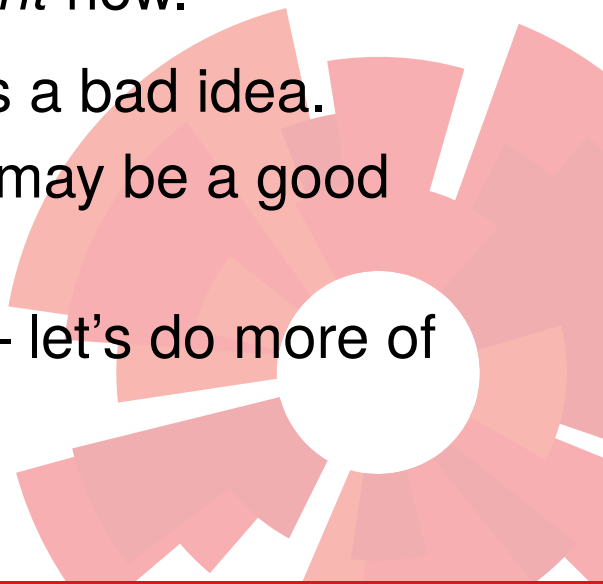
The Zen of Python

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
  - Errors should never pass silently.
  - Unless explicitly silenced.



The Zen of Python

- In the face of ambiguity, refuse the temptation to guess.
- There should be one– and preferably only one –obvious way to do it.
  - Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
  - Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those!



Getting started

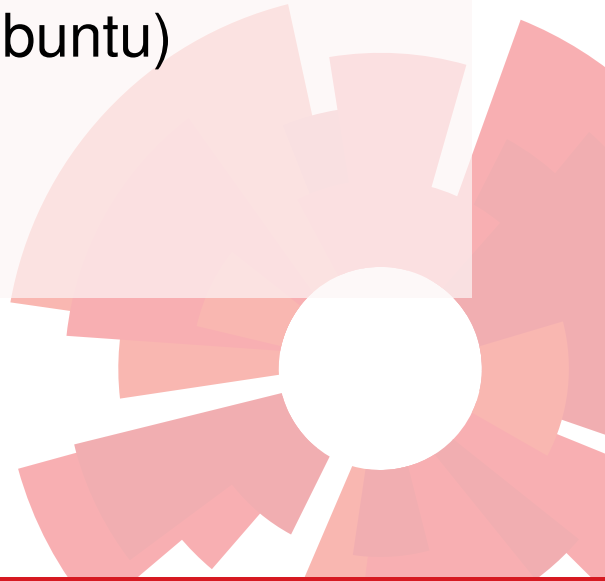
Installing Python 3.6

Windows and Mac OS

- python.org
- or homebrew on Mac OS

Linux

- `apt-get install python3.6` (Debian/Ubuntu)
- `dnf install python3` (Fedora)
- `pacman -S python` (Arch)



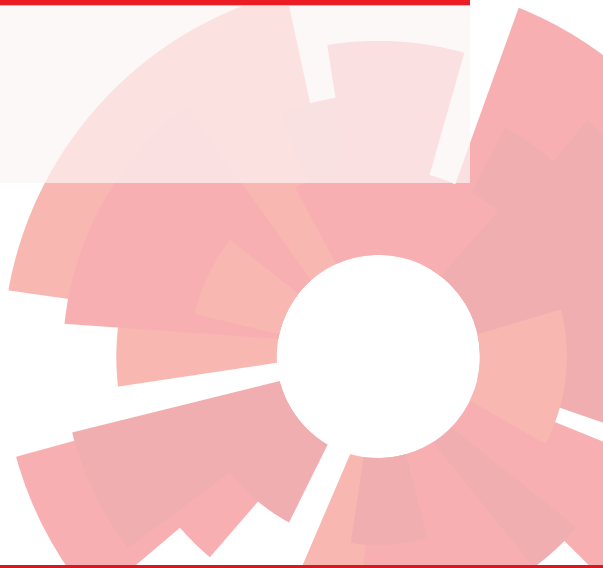
Two “modes” of python

Interactive mode

- Runs Python in a read-evaluate-print-loop (REPL)
- Statements written in an interactive console and evaluated

File mode

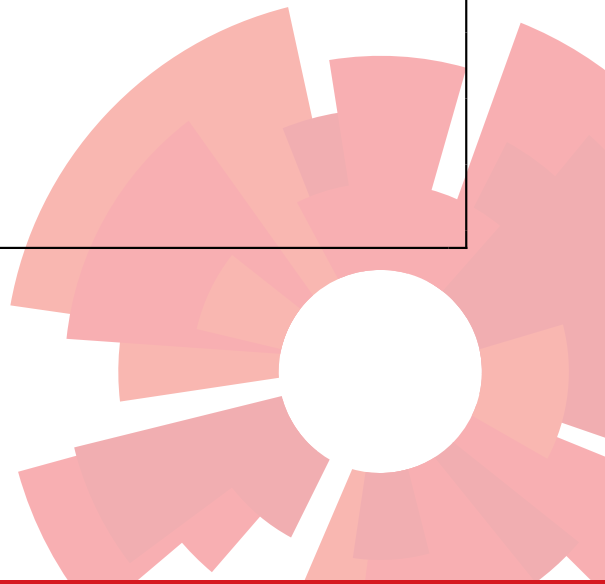
- Runs Python code in a file



Numbers and arithmetic

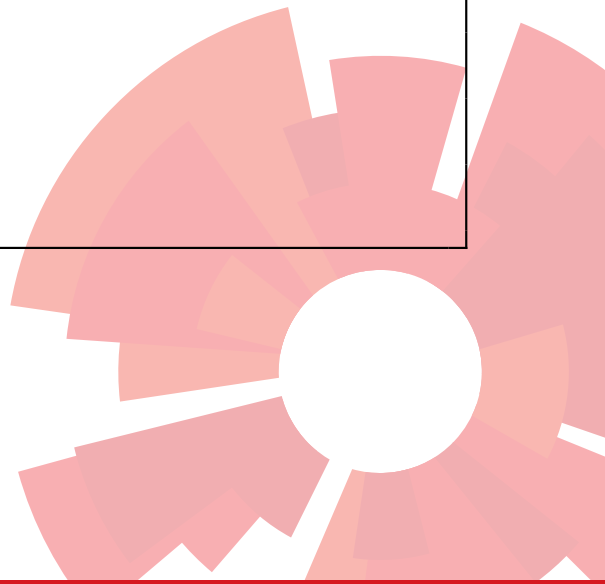
Most things work like you expect

```
>>> 1 + 3 - 4
0
>>> 9 * 12
108
>>> 99 % 5
4
# Division is ALWAYS floating point
>>> 9 / 2
4.5
>>> 9 / 3
3.0
```



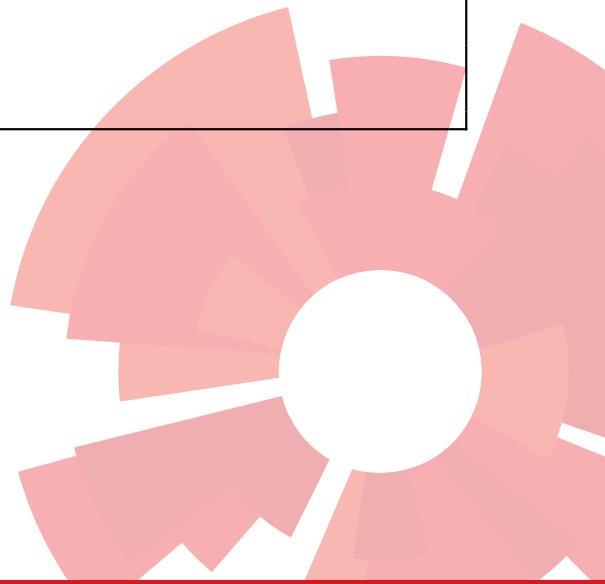
Some things are a bit different

```
# Integer division
>>> 9 // 2
4
>>> 9 // 3
3
# Power
>>> 2 ** 10
1024
# or
>>> pow(2, 10)
1024
```



Comparison

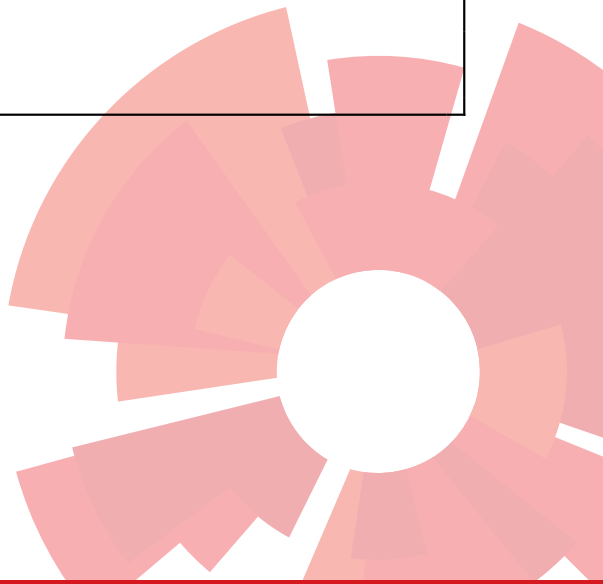
```
# Very familiar
>>> 3 == 2 + 1
True
>>> 12 > 99
False
# Comparison can be chained
>>> 5 < 19 + 3 <= 99
True
```



Integers

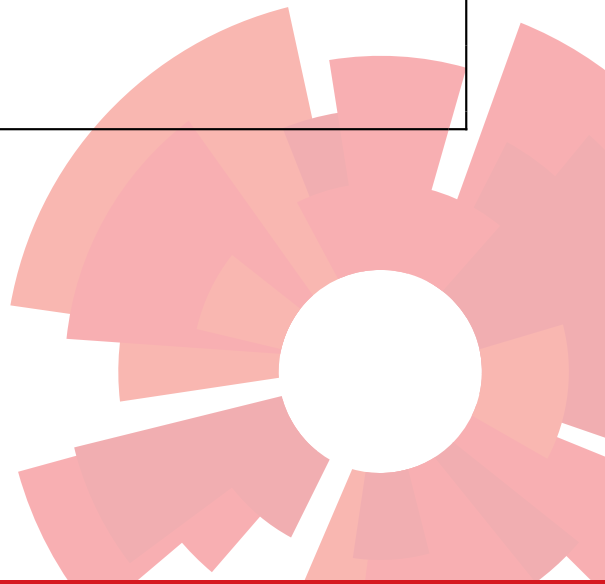
Python uses big integers

```
# No need to worry about overflow
>>> 2 ** 150
1427247692705959881058285969449495136382746624
# Integer literals can use underscores
>>> 1_000_000_000
1000000000
```



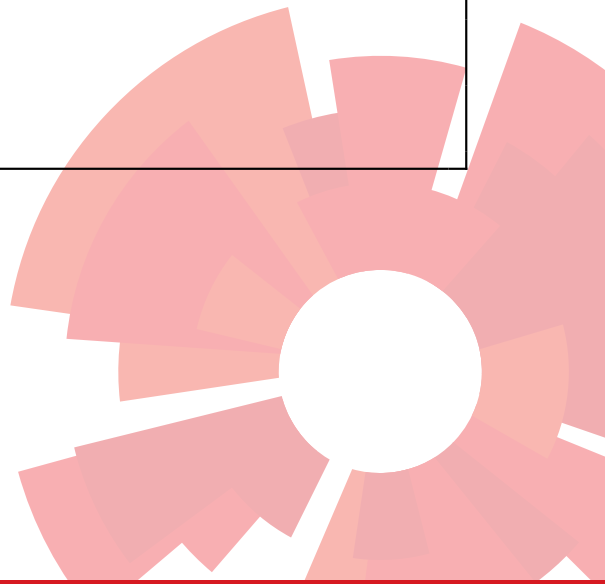
Floating Point Numbers

```
# Your usual syntax works
>>> 93.2312
93.2312
>>> 23.4E23
2.34e+24
# Underscores also work
>>> 1_000_345.5523
1000345.5523
```



Complex Numbers

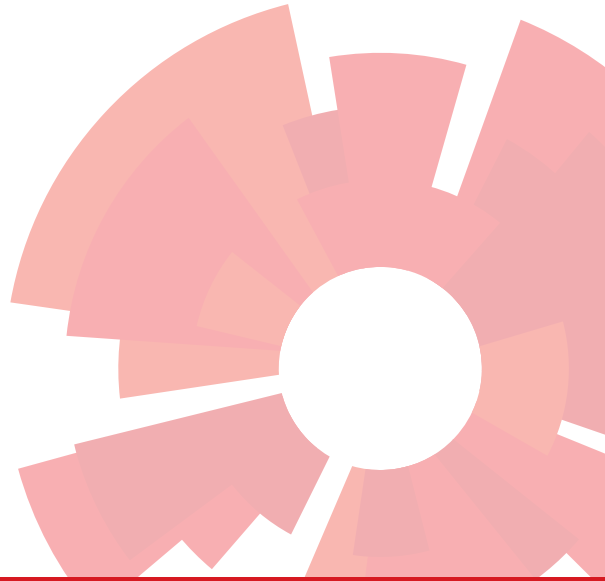
```
# Complex numbers are built in
>>> 1j
1j
>>> 2 + 3j
(2+3j)
>>> (2 + 3j) * (4.3 + 2.1E3j)
(-6291.4+4212.9j)
>>> (2 + 3j) * (2 - 3j)
(13+0j)
```



Strings

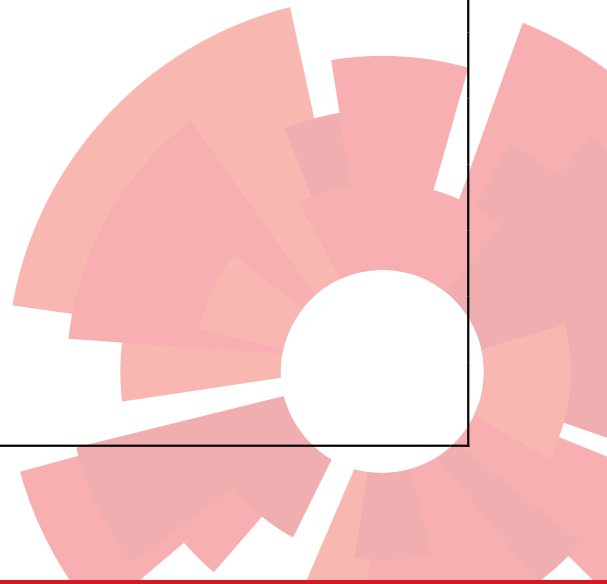
Strings in Python

- Unicode strings*
- “Types” of strings
  - Vanilla strings
  - Multiline strings
  - Raw strings
  - Byte-strings (we’ll cover them later)
  - F-strings (we’ll cover them later)



String syntax

```
'Strings can have either single quotes'
"Or double quotes"

'Python is "fun"'
"Schindler's list"

'''
Multiline strings
are very
very nice
'''

"""
And can also use
double quotes
"""
```



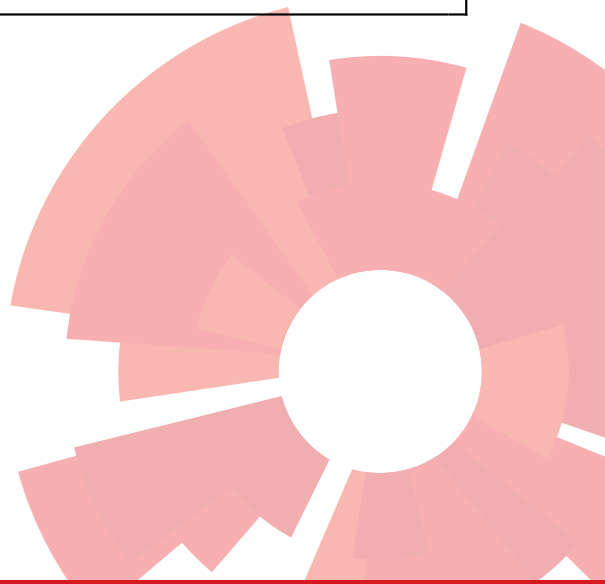
Strings

- Strings in most programming languages use an *escape character* to denote special symbols.
- In Python (like most languages) it is backslash (`\`)
- They are used to denote
  - non-printable characters (`'\a'`)
  - characters (`'\n'`, `'\t'`)
  - syntax escape (`'\"'`, `'\''`, `'\\'`)
- A “character” in Python is just a string of length 1
  - No special construct or syntax

Raw vs. Vanilla

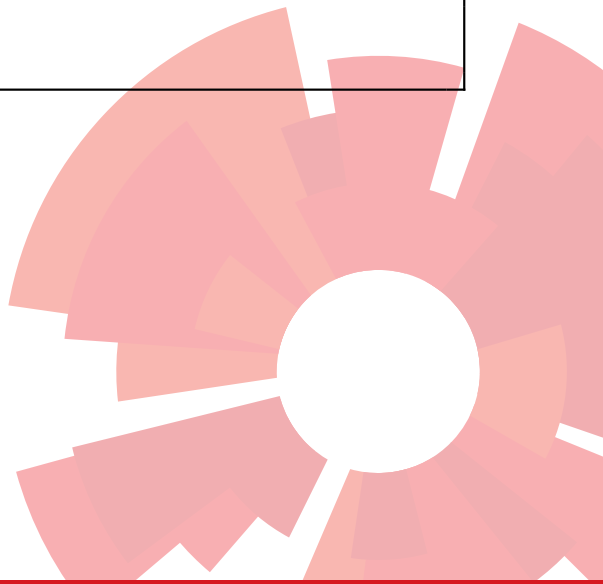
```
# Normal string
>>> 'A new\nline'
'A new\nline'
# Raw strings
>>> r'A new\nline'
'A new\\nline'
```

What's the difference?



Strings

```
# Normal string
>>> print('A new\nline')
A new
line
# Raw strings
>>> print(r'A new\nline')
A new\nline
```



String Operations

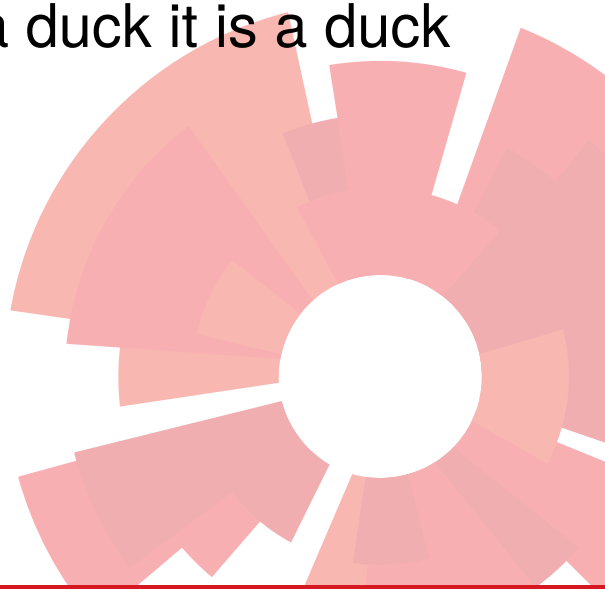
```
# "Size" of string
>>> len('Hello')
5

# String concatenation
>>> 'Hello, ' + 'World!'
'Hello, World!'
# or
>>> 'Hello, ' 'World!'
'Hello, World!'

# Repetition
>>> 19 * 'na' + ' Batman!'
'nanananananananananananananananananananana Batman!'
```

Don’t use string concatenation, we will learn better ways later

Variables

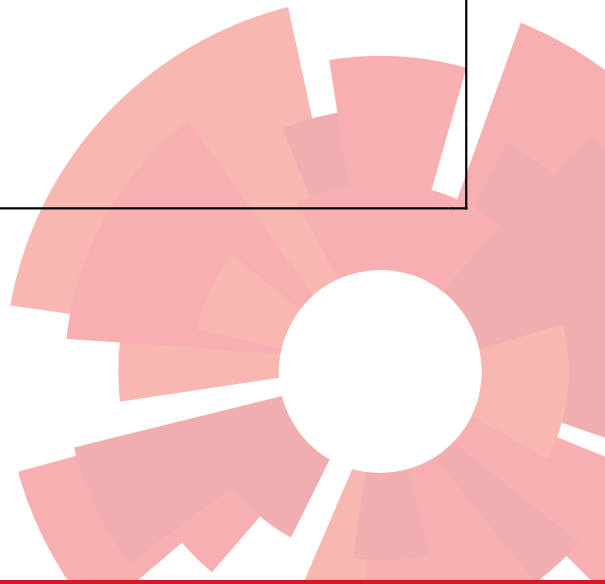
Variables and Types

- Variables in python are dynamically typed
  - We don't specify type in code
  - A variable's type can change at runtime
- "Duck typing"
  - If it looks like a duck and quacks like a duck it is a duck
- Variables are implicitly declared



Example

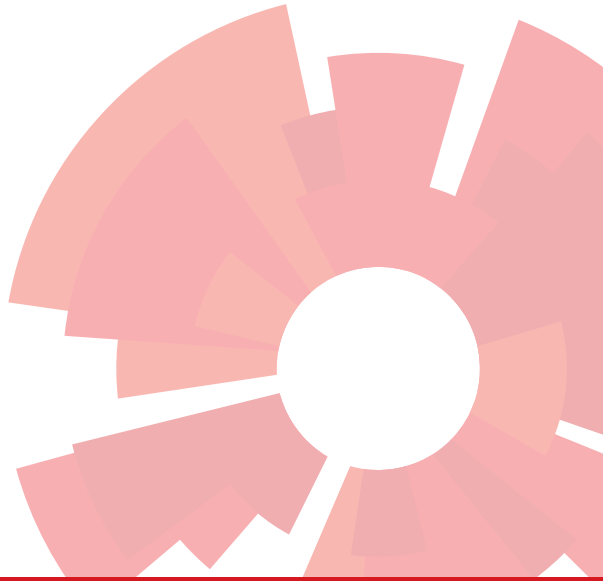
```
# This declares a variable
>>> a = 12
>>> type(a)
<class 'int'>
>>> a + 9
21
# Variables can change types
>>> a = 'Now a is a string'
>>> type(a)
<class 'str'>
```



Lists

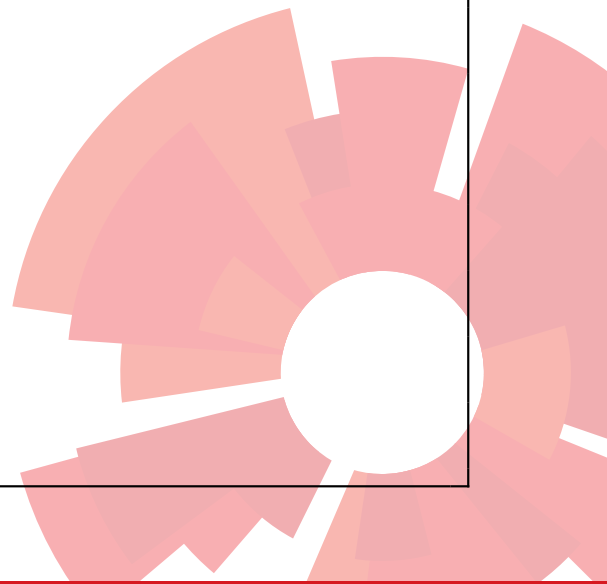
Lists in Python

- Lists are resizable arrays
  - Fast lookup (constant time)
  - Fast append (amortized constant time)
  - Slow insert (linear)



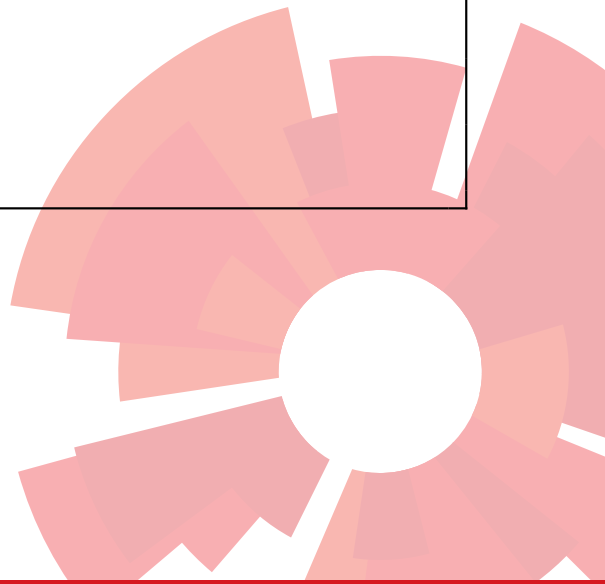
Example

```
# Lists can hold any type
>>> li1 = [1, 2, 3]
>>> li2 = [1, 'a', 3.4]
# Even other lists
>>> li3 = [1, ['a', 3, [3.4]], 'yay']

# Length of lists
>>> len(li2)
3

# List concatenation
>>> [1,3] + [2,4]
[1, 3, 2, 4]

# List repetition
>>> 4 * [1,2]
[1, 2, 1, 2, 1, 2, 1, 2]
```



Member access

```
# Member access
>>> a = [1, 'yay', [5,3]]
>>> a[2]
[5, 3]
>>> a[0]
1
>>> a[1][2]
'y'
>>> a[2][0]
5
```

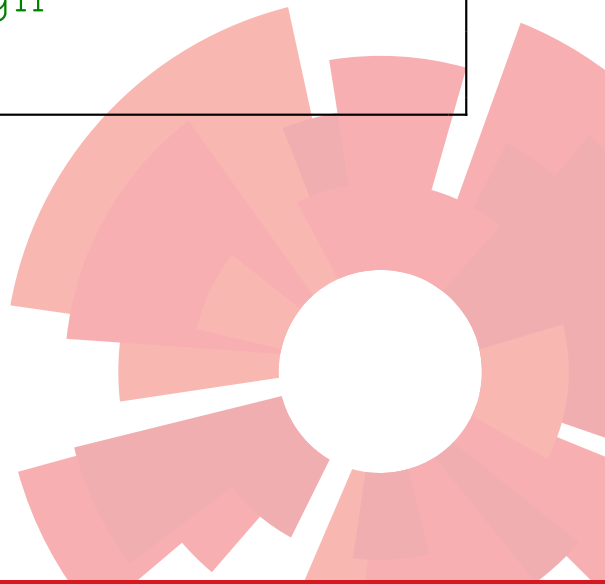


Booleans

True and false

`True` and `False` are synonyms for 1 and 0

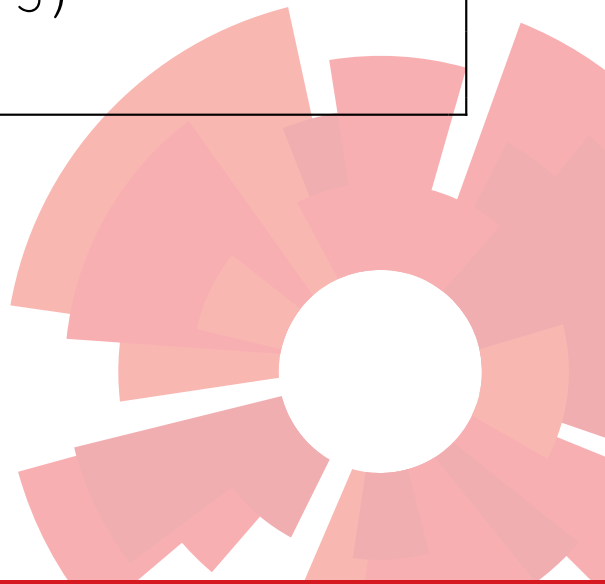
```
>>> True == 1
True
>>> False == 0
True
>>> True + 3     # Don't do this, though
4
```



Boolean Operators

The boolean operators are **and**, **or** and **not**

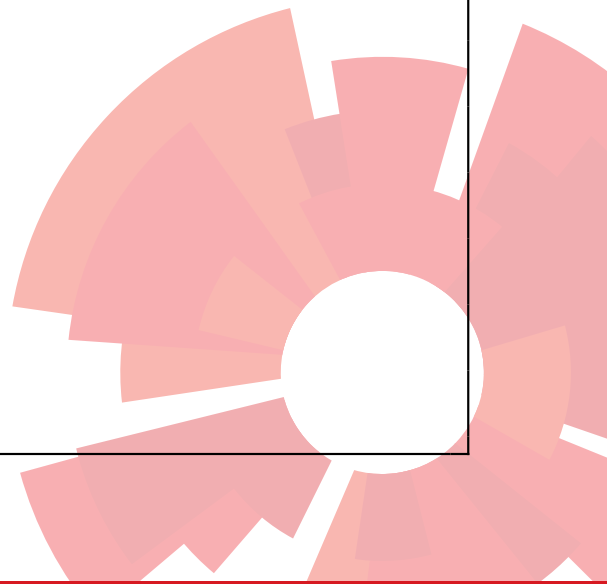
```
>>> True or False
True
>>> 1 < 3 and 4 > 2
True
>>> 9 * 2 < 12 and not (3 < 2 or 1 == 5)
False
```



Boolean Operators

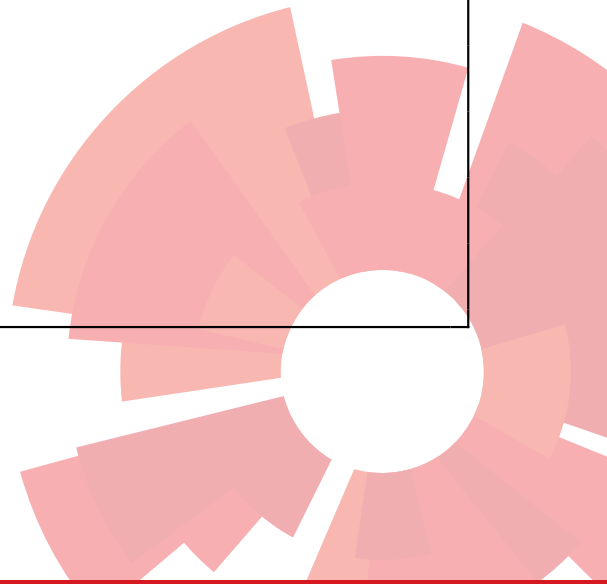
- The boolean operators are smart
  - Built in function **`bool`** is applied to their parameters

```
>>> bool(0)
False
>>> bool(1)
True
>>> bool([])
False
>>> bool([1, 2, 3])
True
>>> bool('')
False
>>> bool('hello')
True
```



Boolean Operators with Objects

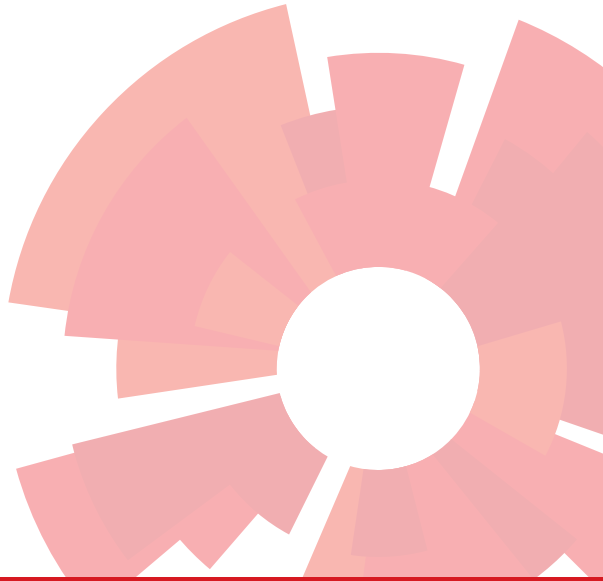
```
>>> [] or 0 or 'yay'
'yay'
>>> [] or 5 or 'yay'
5
>>> [] or 0 or ''
''

>>> [1,2,3] and ''
''
>>> [1,2,3] and not ''
True
>>> [1,2,3] and 'w00t'
'w00t'
```



Objects

Everything is an object

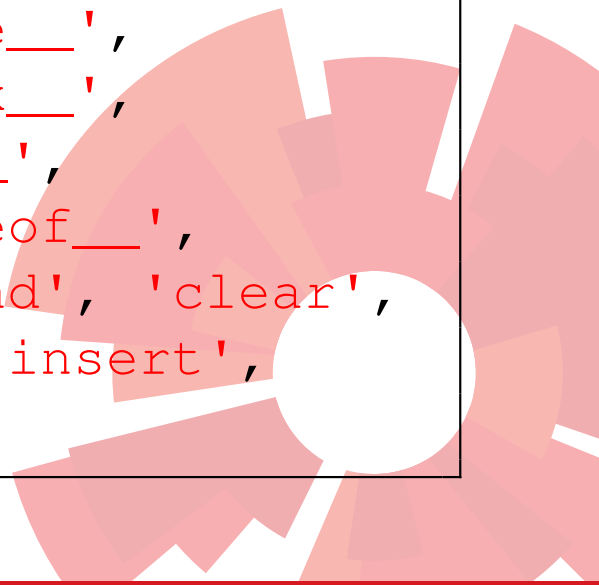
- Numbers are objects
- Functions are objects
- Classes are objects
- Modules are objects
- `is` vs `==`



dir

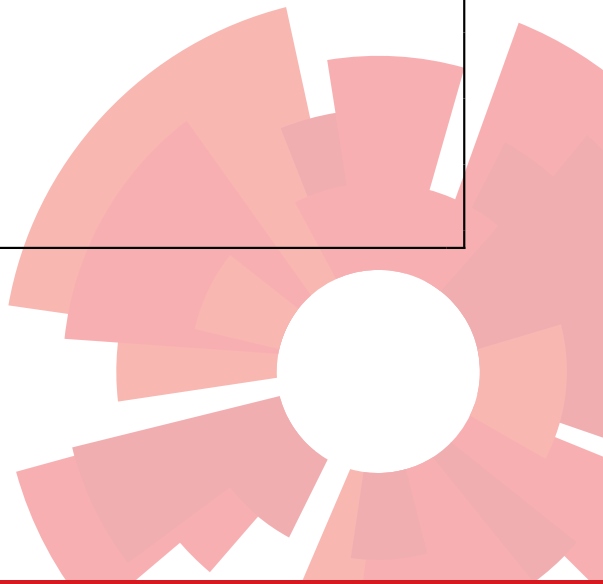
To show all members of an object you can use the built in function **dir**

```
>>> dir([])
['__add__', '__class__', '__contains__',
 '__delattr__', '__delitem__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__getitem__', '__gt__',
 '__hash__', '__iadd__', '__imul__', '__init__',
 '__init_subclass__', '__iter__', '__le__',
 '__len__', '__lt__', '__mul__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__rmul__',
 '__setattr__', '__setitem__', '__sizeof__',
 '__str__', '__subclasshook__', 'append', 'clear',
 'copy', 'count', 'extend', 'index', 'insert',
 'pop', 'remove', 'reverse', 'sort']
```



Objects

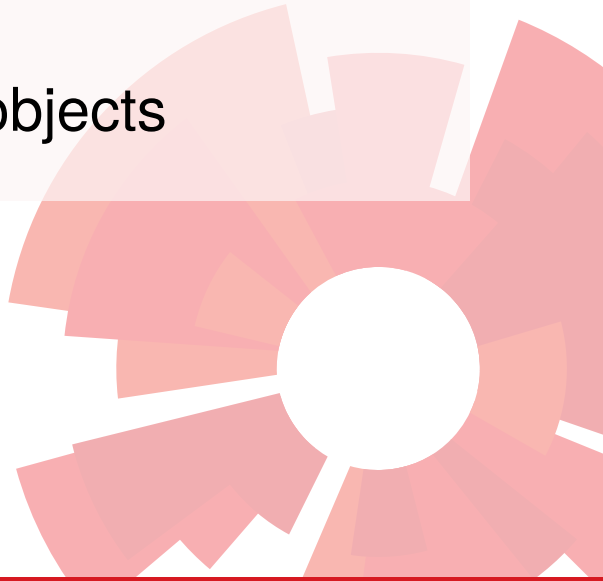
```
>>> type(5)
<class 'int'>
>>> (5).__add__(3)       # Same as 5 + 3
8

>>> type(print)
<class 'builtin_function_or_method'>
>>> type(type)
<class 'type'>
>>> type([])
<class 'list'>
```



Object comparison

We have two ways of comparing objects

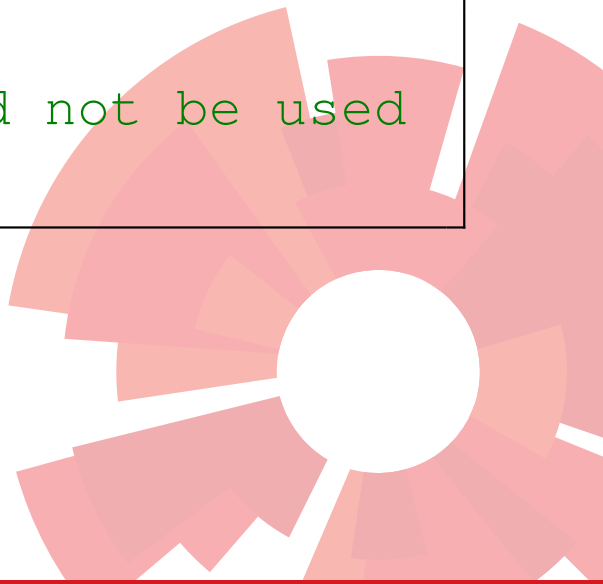
- `is` checks if two objects are **the same exact object**
  - Used to check for `None`
  - Otherwise, this is very rarely what we want to do!
- == checks if two objects are *equal*
  - We can define what it means for our objects



None

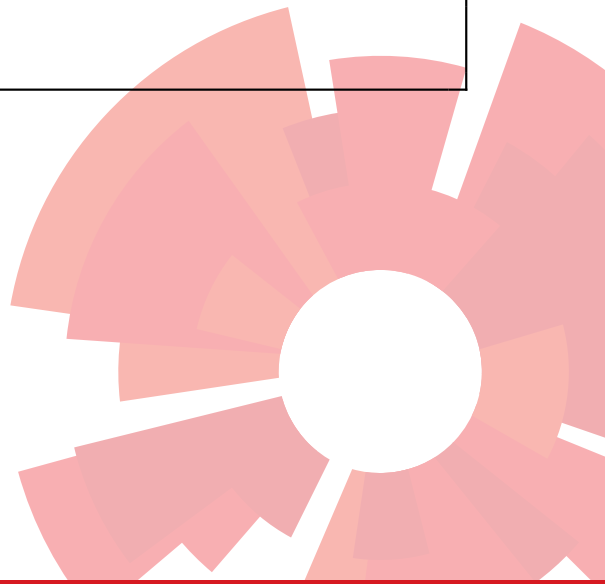
- `None` is a special object to denote no value
- Same concept as `null` in other languages
  - But not exactly the same!

```
>>> a = None
>>> a is None
True
>>> a == None  # This works but should not be used
True
```



Object comparison

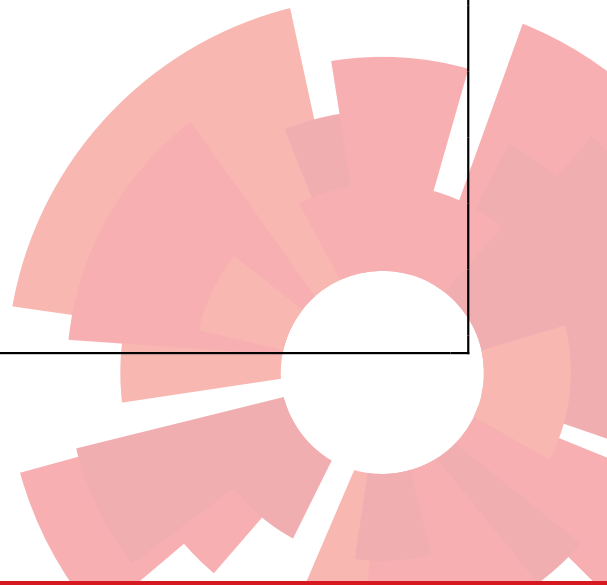
```
>>> a = 3
>>> b = 3.0

>>> a == b
True
>>> a is b
False
```



Is is dangerous

`is` should, e.g., not be used with numbers

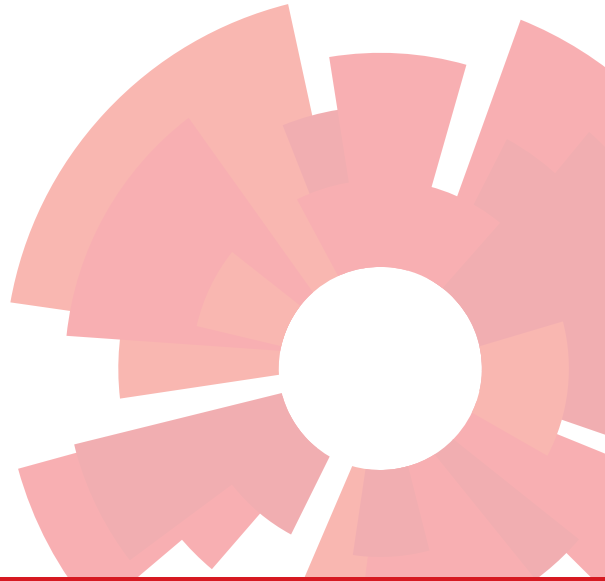
```
>>> x = 3
>>> y = 3
>>> x is y
True

>>> 123123123 is 123123123
True

>>> a = 123123123
>>> b = 123123123
>>> a is b
False
```



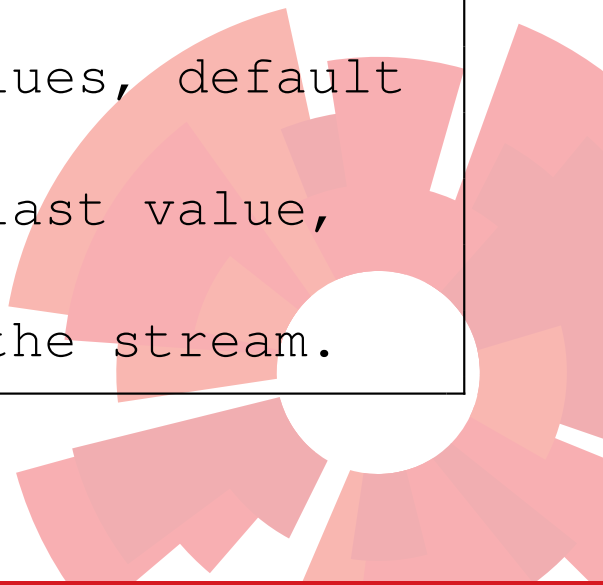
Documentation in the REPL

Help

- `dir` shows all members of an object
- `help` shows documentation of an object



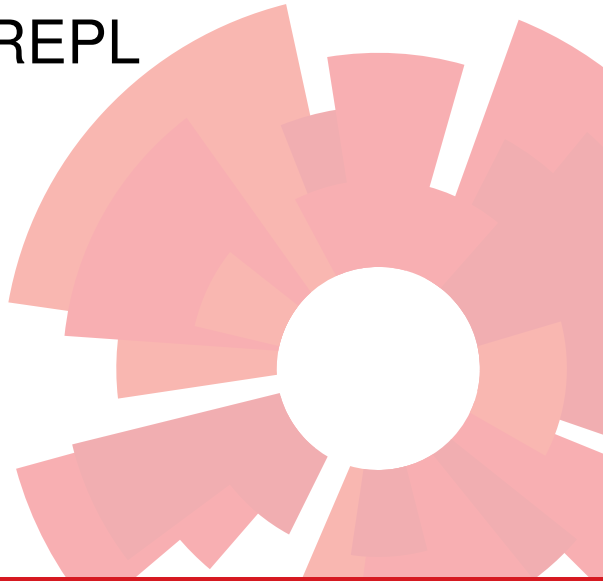
```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.
        stdout, flush=False)

    Prints the values to a stream, or to sys.stdout
        by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to
        the current sys.stdout.
    sep:   string inserted between values, default
        a space.
    end:   string appended after the last value,
        default a newline.
    flush: whether to forcibly flush the stream.
```



Moving to files

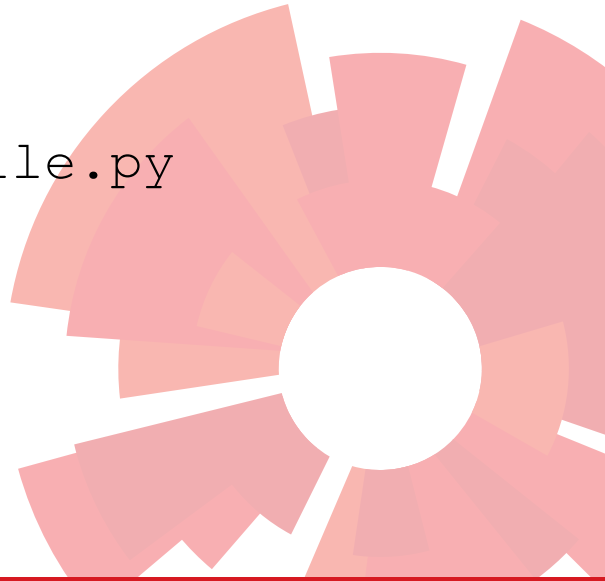
Running python as a script

- Python code can be written to a file and run
  - It does not require (explicit) compilation
  - In this case we usually call the file a script (or a program)
- Also possible to run script and then enter REPL



Windows

- GUI method
  - Double-click on python file
  - A python terminal will open while the script is running and close as soon as it stops
- Command line method
  - Open cmd/powershell
  - `C:\...> py file.py` or `C:\...> file.py`



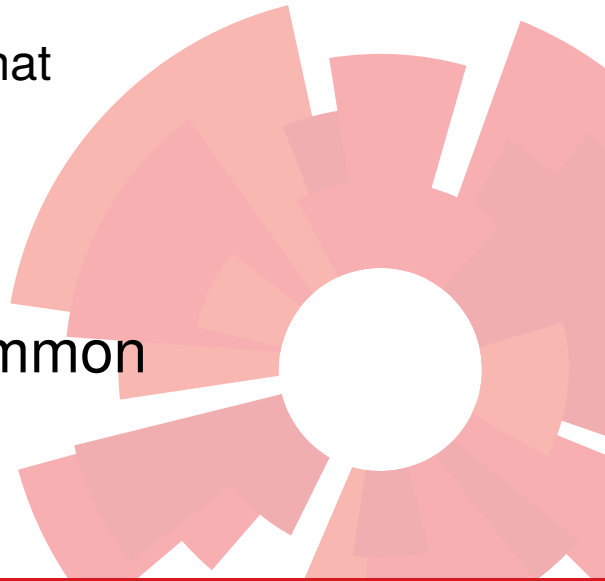
Linux/MAC OS

- Run with Python
  - Open a terminal
  - `$ python3 file.py`
- Shebang method
  - Put the line `#! /usr/bin/env python3` at the top of `file.py`
  - Run `$ chmod +x file.py` or `$ chmod u+x file.py`
  - To run Python script run `$ ./file.py`

Control structures

Indentation

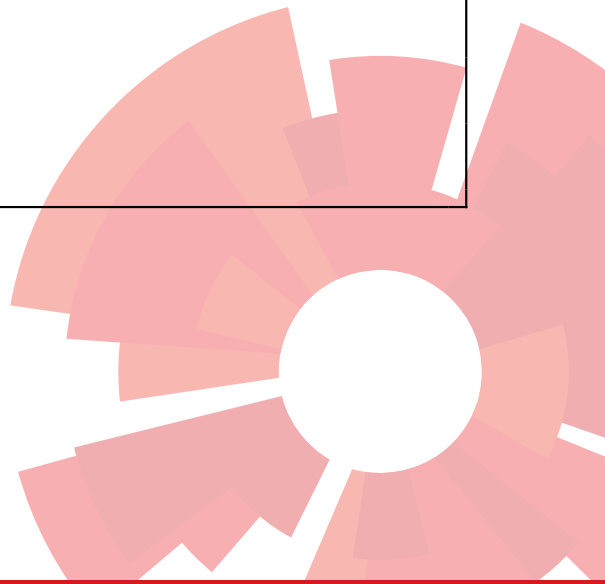
- Python does not use braces to denote body of control structures, functions, etc.
- Indentation is used in stead
  - All lines in the same block must have the same indentation
  - Indentation does not need to be consistent between blocks
    - It's a terrible idea to take advantage of that
  - Tabs and spaces can both be used
    - But don't use tabs
  - Indentation of four spaces is most common



If statements

```
a = 99
if a < 3:
    print('a is small')
    if a < 0:
        print('a is negative')
    else:
        pass    # A no-op statement
elif 3 <= a <= 10:
    print('a is slightly larger')
else:
    print('a is big')
```

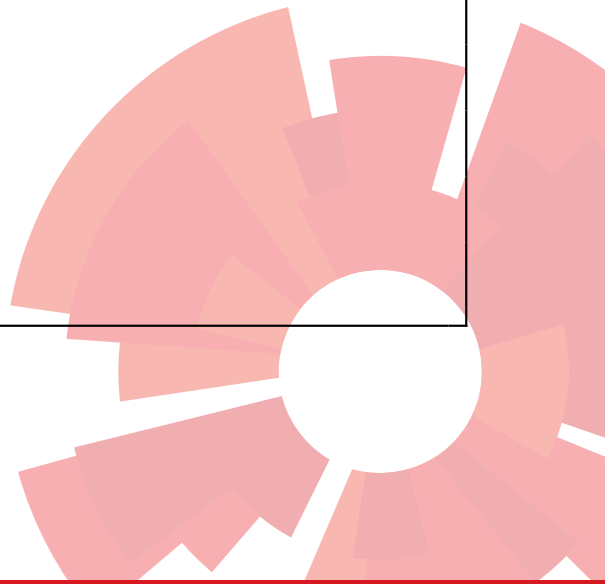
if.py



Indentation again

```
if condition:
    a = 'This is the normal indentation'
else:
  b = 'This is allowed' # but a terrible idea!
  if another_condition:
   c = 'Even this is allowed'
    d = 'This is a syntax error'
   e = "I'm still in the if-statement"
  f = "I'm in the else-statement"

# This is also a syntax error
# Blocks cannot be empty
if condition:
    # TODO: Implement later
```

indentation.py

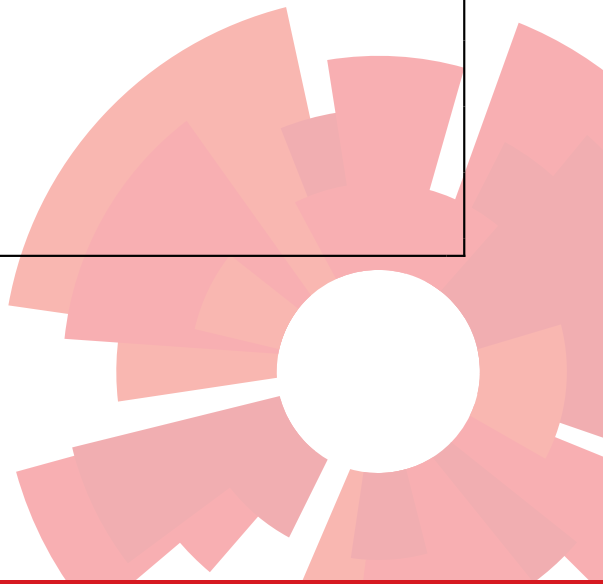


While

- Similar to C++/Java/...
- **break** and **continue** work as expected
- Not used a lot in Python

```
num = 0

while num < 10:
    if num == 5:
        continue
    print('num is now', num)
    num -= 1
```

while.py

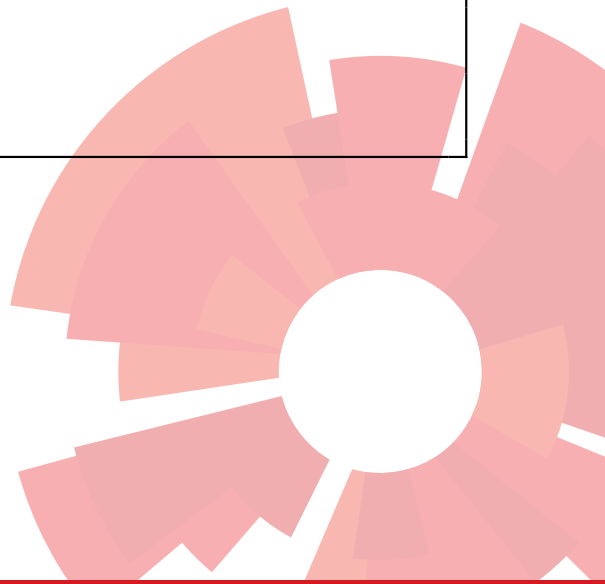


For

- For-each loops, used to iterate over collections
  - Considered more idiomatic python (more “pythonic”)
- Collection must be *iterable*

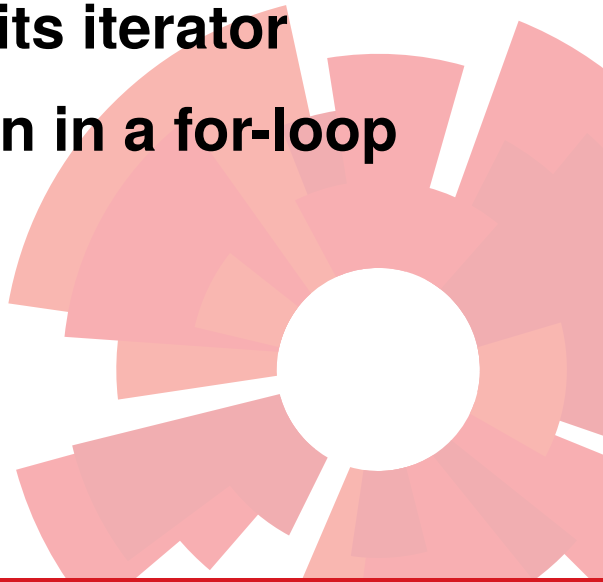
```
lis = [1, 5, 2, 3, 4]

for i in lis:
    print('i is now', i)
```

for.py



Iterators

- Iterators provide access to a collection's members in some order
- The built-in function `iter` creates an iterator and `next` *consumes* it
- We rarely use iterators explicitly
- **Do not modify a collection when using its iterator**
  - Therefore, **do not modify a collection in a for-loop**

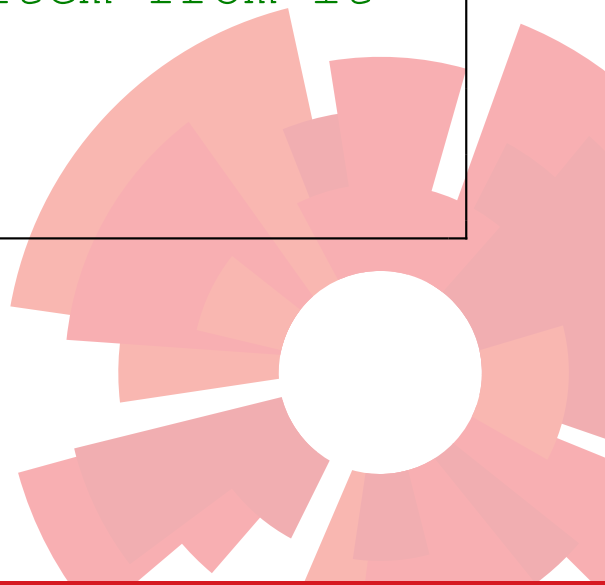


For-loops and iterators

This is roughly the equivalent of a for-loop implemented with a while-loop

```
lis = [1, 5, 2, 3, 4]

it = iter(lis)    # Create iterator
while True:
    try:
        i = next(it)    # Consume one item from it
    except StopIteration:
        break
    print('i is now', i)
```

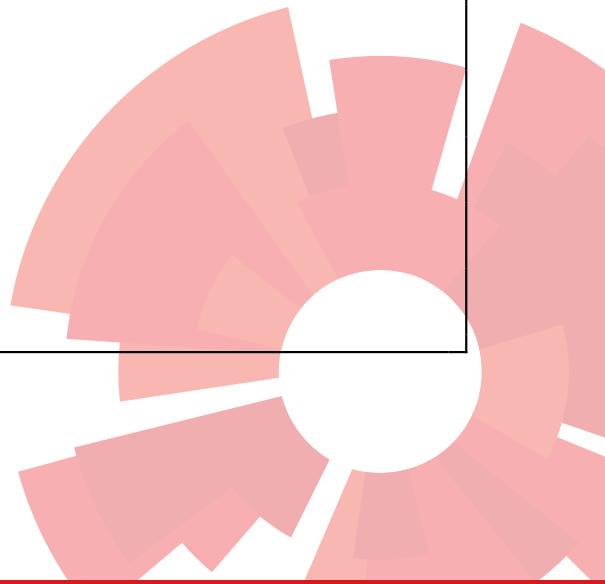
for-iter.py



Range

To loop over a range of number we use **range**

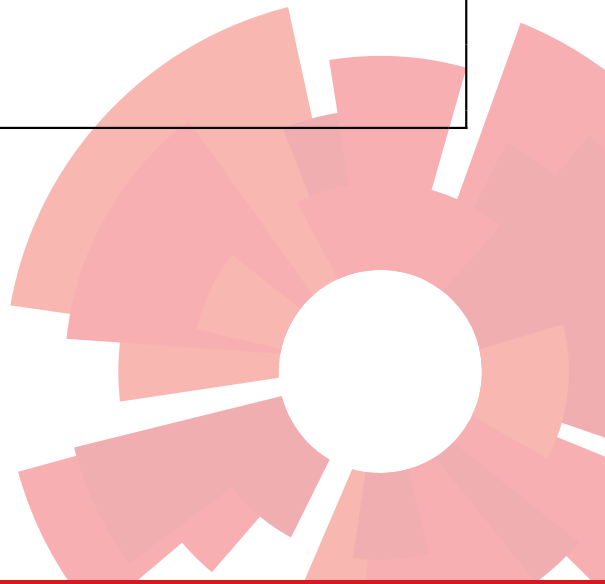
```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> r = range(3)
>>> it = iter(r)
>>> next(it)
0
>>> next(it)
1
>>> next(it)
2
```



For with range

```
for i in range(10):
    print('i is now', i)


for i in range(2, 10):
    print('i is now', i)

for i in range(2, 10, 2):
    print('i is now', i)
```

for-range.py

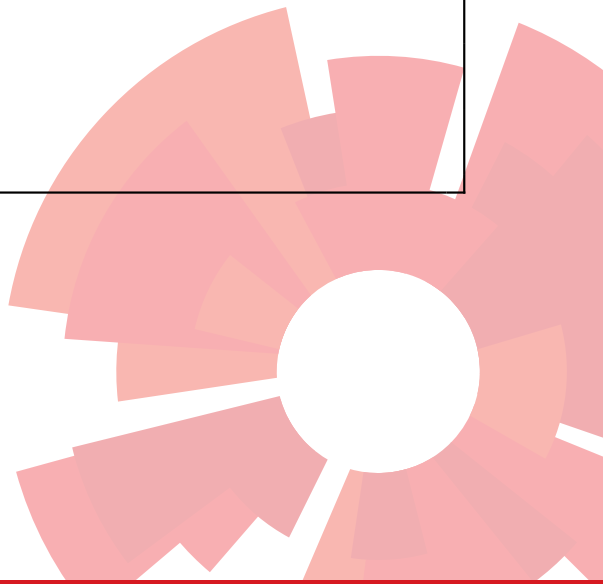


Lists and range

Do not use `range` with loops to access members of a list

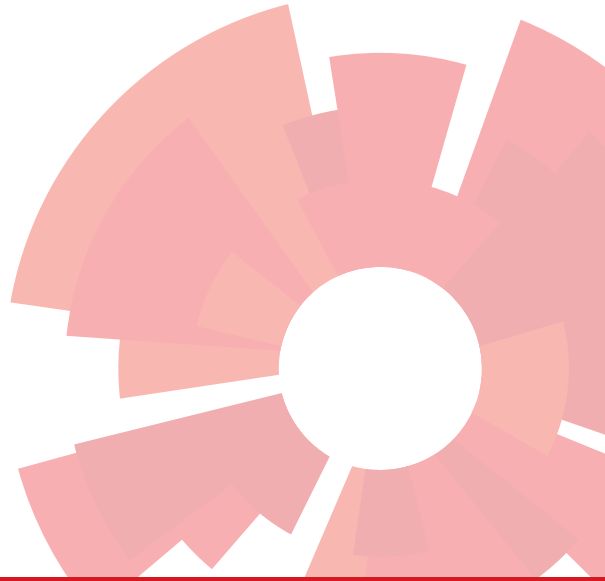
```
lis = [1, 5, 2, 3, 4]

# Don't do!
for i in range(len(lis)):
    print('i is now', lis[i])

# Do do
for i in lis:
    print('i is now', i)
```

for-range-bad.py



More on looping

- Indices
  - enumerate
- Two (or more) lists simultaneously
  - zip

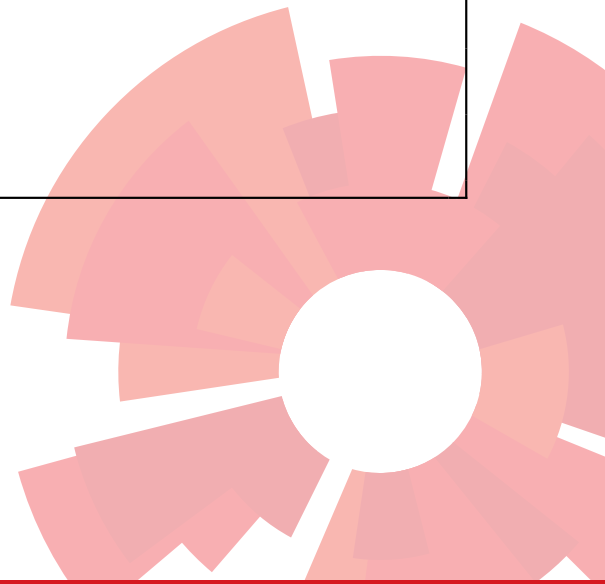


for-else (while-else)

`else` clauses for loops are executed if the loop's execution was **not** stopped by a `break`

```
num = 44

for i in range(2,num):
    if num % i == 0:
        print(num, 'is not a prime')
        break
else:
    print(num, 'is a prime')
```

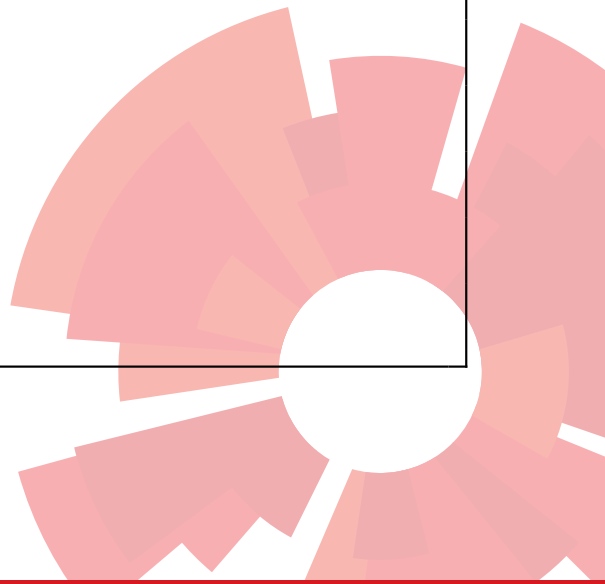
for-else.py



Functions

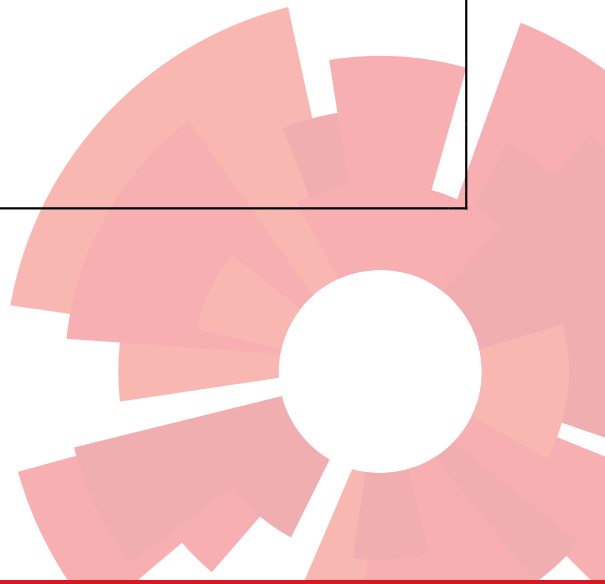
Definition

```
>>> def foo():
...     return 'Hello, World!'
>>> foo()
'Hello, World!'

>>> def bar():
...     pass
# No return statement => function returns None
>>> bar()
>>> print(bar())
None

>>> type(foo)
<class 'function'>
```



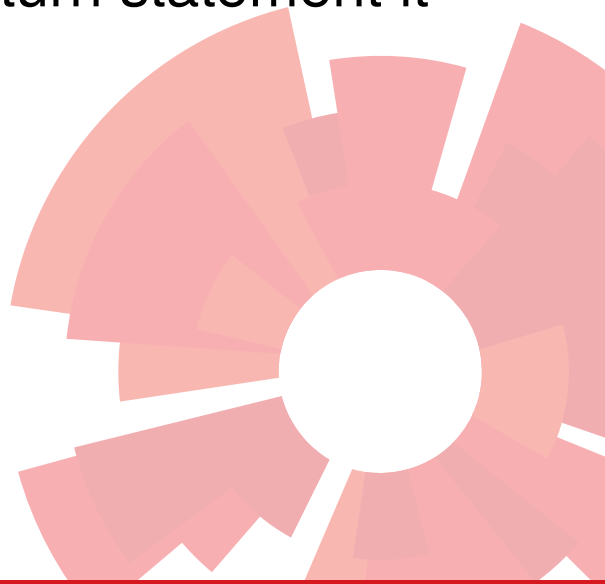
Duck typing

```
>>> def fun(a, b):
...     return a + b
>>> fun(1, 3)
4
>>> fun(1.4, 3.6)
5.0
>>> fun('Hello, ', 'World!')
'Hello, World!'
>>> fun([1, 2], [4, 3])
[1, 2, 4, 3]
```



More about functions

- Return types do not have to be consistent
  - But that, of course, is a bad idea
- All functions have a return value
  - If function ends in a branch with no return statement it returns `None`

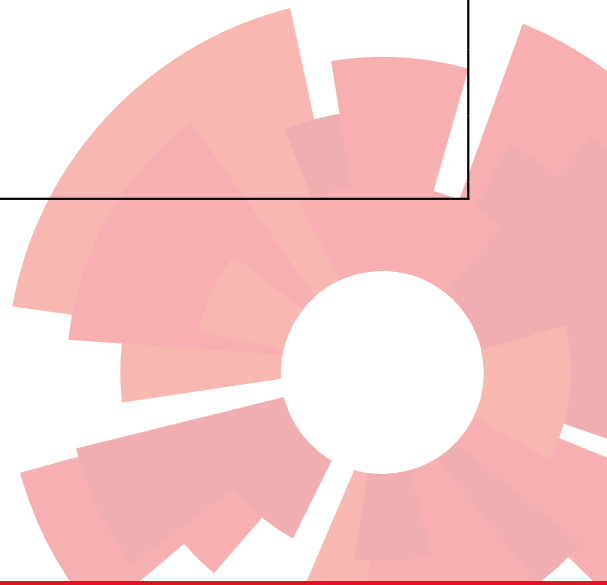


Docstrings

If the first statement of a function is a string, it's considered that function's documentation

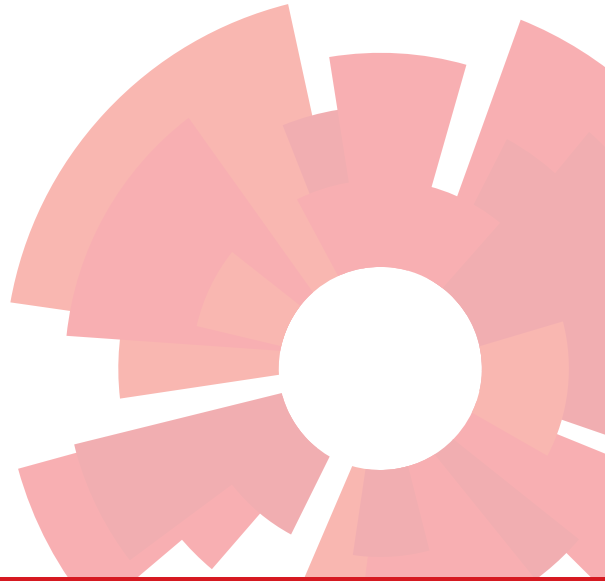
```
def fun(a, b):
    """
    Returns the sum of 'a' and 'b'
    If 'a' or 'b' is None, returns None
    """
    if a is None or b is None:
        return None
    return a + b
```

docstring.py



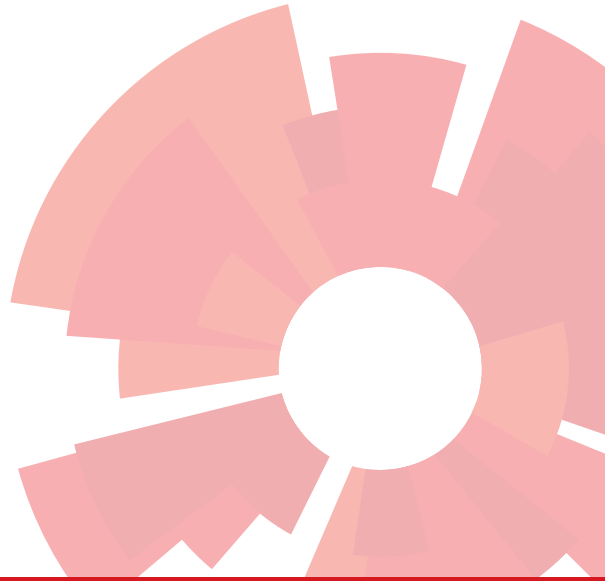
Function parameters

- Positional (aka normal arguments)
- Named (with default value)
- Variable-length arguments
- Variable-length keyword arguments



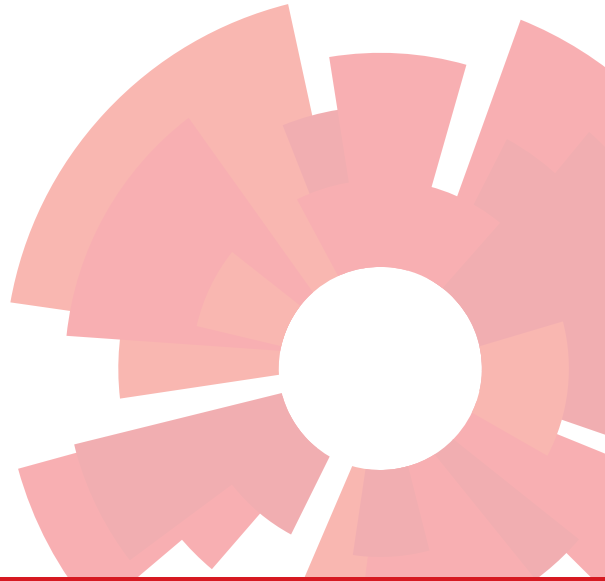
Variable-length arguments

```
>>> def varlen(*args):
...     print(args)
>>> varlen(1, 2, 3, 'a', 'b', 'c')
(1, 2, 3, 'a', 'b', 'c')
```



Variable-length keyword arguments

```
>>> def varkey(**args):
...     print(args)
>>> varkey(a=1, b=2, c=3, d='a', e='b')
{'a': 1, 'b': 2, 'c': 3, 'd': 'a', 'e': 'b'}
```



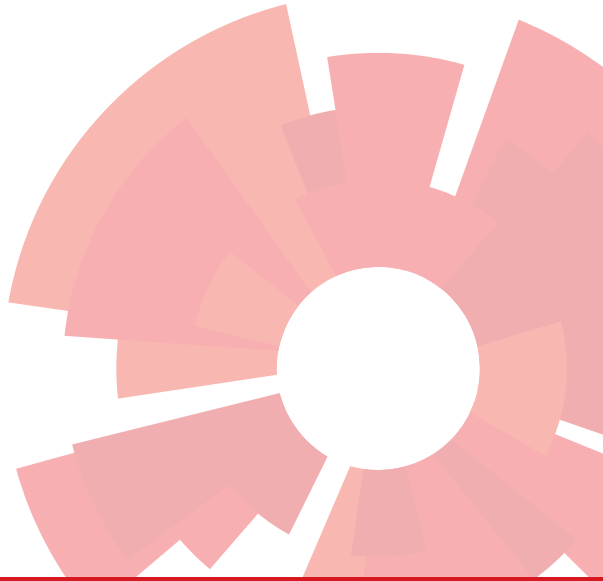
Scope

Global scope

- Variables declared outside of a function are global (in that module)
  - Accessible everywhere (within module)
- `globals()` – built in function that returns all variables in global scope
- `global` – keyword that declares a variable to be in global scope

Local scope

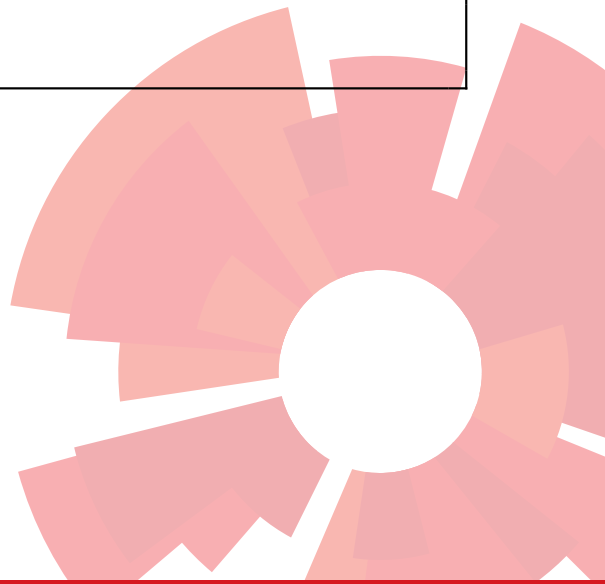
- Only created by functions
  - **Not** created by blocks (like in C++, Java, . . .)
- `locals()` – built in function that returns all variables in global scope
- Inherited by nested functions
- Shadows global scope
- Closures



Common mistake

```
glob = 3    # This variable is in global scope

def fun():
    print(glob)
    # Declares a new variable glob in fun's scope
    glob += 3

fun()
```

scope-error.py

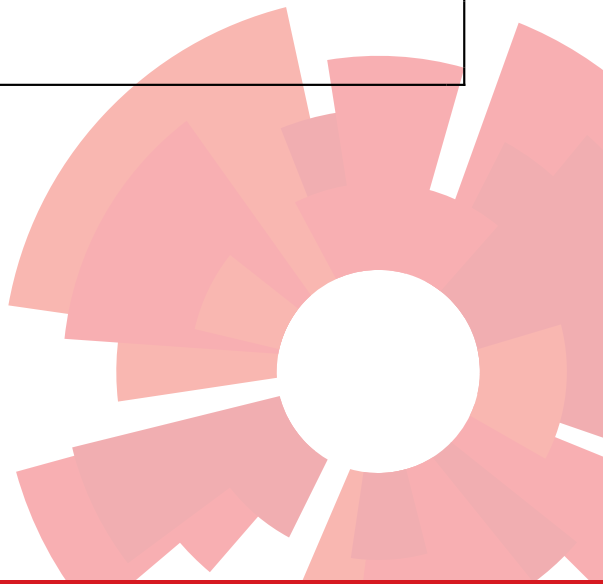


Common mistake fixed

```
glob = 3    # This variable is in global scope

def fun():
    # Tells Python to use glob from global scope
    global glob
    print(glob)
    glob += 3

fun()
```

scope-fixed.py

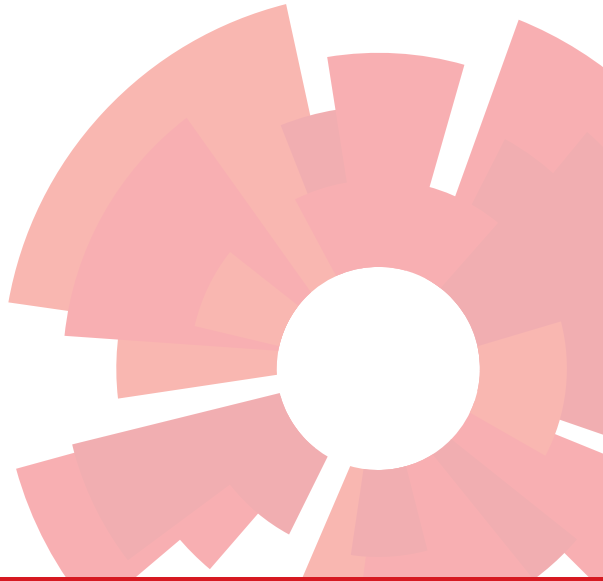
However, don't use the `global` keyword



Mooshak

Errors

- Compile time error
  - Wrong syntax
  - Wrong function name
- Wrong answer
- Runtime error
- Time limit exceeded



Input, output and diff

- Parameter syntax
- Diff
- Diff/input not always shown