- [1] Building Abstractions with functions
- 1.1 Getting Started
- Computer fundamentals:

representing information

specifying logic to process it

designing abstractions that manage the complexity of that topic

- Recommended textbook: Structure and Interpretation of Computer Programs (SICP)
- 1.1.1 Programming in Python
- 1.1.2 Installing Python 3
- 1.1.3 Interactive Sessions
- In an interactive Python session, you type some Python code after the ">>>". The Python interpreter reads and executes what you type, carrying out your various commands.
- To start an interactive session, type "python3" at a terminal prompt.
- Interactive controls: each session keeps a history of what you have typed. To access that history, press [Ctrl-p] (previous) or [Ctrl-n] (next). You can exit a session with [Ctrl-d], which discards the history. Up and down arrows also cycle through history on some systems.
- 1.1.4 First Example
- Statements & Expressions: Python code consists of expressions and statements. Broadly, computer programs consist of instructions to either computer some value or carry out some actions.

  Statements typically describe actions. Expressions typically describe computations.
- Functions: functions encapsulates logic that manipulates data.
- Objects: an object seamlessly bundles together data and the logic that manipulates that data, in a way that manages the complexity of both.
- Interpreters: evaluating compound expressions requires a precise procedure that interprets code in a predictable way. A program that implements such a procedure, evaluating compound expressions, is called an interpreter. When compared with other computer programs, interpreters for programming languages are unique in their generality.
- 1.1.5 Errors
- Computers are rigid: even the smallest spelling and formatting changes can cause unexpected output and errors. Learning to interpret errors and diagnose the cause of unexpected errors is called debugging.
- Some guiding principles of debugging are:

Test incrementally

Isolate errors: trace the error to the smallest fragment of code you can before correcting it. Check your assumptions

Consult others

## 1.2 Elements of Programming

- Programs serve to communicate ideas among members for a programming community. Thus,
   programs must be written for people to read, and only incidentally for machines to execute.
- When we describe a language, we should pay particular attention to the means that the language provide for combining simple ideas to from more complex ideas. Each powerful language has three such mechanisms:

primitive expressions and statements: represent the simplest building blocks

means of combination: by which compound elements are built from simpler ones means of abstraction: by which compound elements can be named and manipulated as units

## 1.2.1 Expressions

- primitive expressions like numbers in base 10: 42
- compound expressions like numbers combined with mathematical operators: -1 -1

## 1.2.2 Call Expressions

- The most important kind of compound expression is a call expression, which applies a function to
- A call expression can have subexpressions.
- The order of the arguments in a call function matters.
- Function notation has three principle advantages over the mathematical convention of infix notation

functions may take an arbitrary number of arguments

function notation extends in a straightforward way to nested expressions

mathematical notations has a great variety of forms, while this complexity can be unified via the notation of call expressions

## 1.2.3 Importing Library Functions

• Python defines a very large number of functions, including the operator functions mentioned in the preceding section, but does not make all of their names available by default. Instead, it organizes the functions and other quantities that it knows into modules,, which together comprise the Python library. To use these elements, one imports them. Below are examples:

>>> from math import sqrt

>>> from operator import add, mul

## 1.2.4 Names and the Environment

- A critical aspect of a programming language it the means it provides for using names to refer to computational objects. If a value as been given a name, we say that the name binds to the value.
- In Python, we can establish new bindings using the assignment statement, like

```
>>> radius = 10
```

>>> radius

10

Here "=" is called the assignment operator. <mark>Assignment is our simplest means of abstraction</mark>.

- The possibility of binding names to values and later retrieving those values by names means that the interpreter must maintain some sort of memory that keeps track of the names, values and bindings. This memory is called an environment.
- Names can also be bound to functions, like

```
>>> f = max
>>> f
```

<built-in function max>

>>> f(1, 2, 3, 4)

 In Python, names are called variable names or variables, because they can be bound to different values in the course of executing a program. When a name is bound to a new value through assignment, it is no longer bound to any previous value. One can even bind built-in names to new values, like max.

We can assign multiple values to multiple names in a single statement, like
 area, circumference = pi \* radius \* radius, 2 \* pi \* radius
 note that after assignment, even if we change the value of radius, area will not change

(because in nature, variables are bound to values, not expressions)

- 1.2.5 Evaluating Nested Expressions
- To evaluate a call expression, Python will do the following: evaluate the operator and operand subexpressions

apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions

- Note that the evaluation procedure is recursive in nature. If we draw each expression that we evaluate, we can visualize the hierarchical structure of this process. This illustrations is called an expression tree. In computer science, trees conventionally grow from the top down. The objects at each point in a tree are called nodes, here they are expressions paired with values.
- 1.2.6 The Non-Pure Print Function
- Pure functions: have some input (arguments) and return some output (the result of applying them). Pure functions have the property that applying them has no effects beyond returning a value. Moreover, a pure function must always return the same value when passed in same arguments.
- Non-pure function: can generate side effects, which make some changes to the state of the
  interpreter or computer. A common side effect is to generate additional output beyond the return
  value, using the print function. The value that print returns is always None (a special Python
  value that represents nothing)
- Below is a special case:

>>>print(print(1), print(2))

1

2

None, None

#Note that None evaluates to nothing, and will not be displayed by the interpreter as a value, like >>> None

## 1.3 Defining New Functions

- 3 elements which make Python and other programming languages so powerful:

  Numbers and arithmetic operations are primitive built-in data values and functions

  Nested function application provides a means of combining operations

  Binding names to values provides a limited means of abstraction
- Function definition is a much more powerful abstraction technique by which a name can be bound to compound operation, which can then be referred to as a unit.
- How to define a function:

def <name> (<formal parameters>):

return <return expression>

The return expression is not evaluated right away; it is stored as part of the newly defined function and evaluated only when the function is eventually applied.

 We can use the defined function as a building block in defining other function. Actually, userdefined functions are used in exactly the same way as built-in functions

#### 1.3.1 Environments

- An environment in which an expression is evaluated consists of a sequence of frames, depicted as
   boxes. Each frame contains bindings, each of which associates a name with its corresponding
   value.
- global frame
- The name of a function can be repeated twice, once in the frame and again as part of the function itself. The name appearing in the function is called the intrinsic name. The name in a frame is a bound name. There is a difference between the two: different names may refer to the same function, but that function itself has only one intrinsic name. The name bound to a function in a frame is the one used during evaluation. The intrinsic name of a function does not play a role
- Function signature: a description of the formal parameters of a function, like max(...). Note that the "..." here means max can take an arbitrary number of arguments.
- 1.3.2 Calling User-Defined Functions
- Applying a user-defined function introduces a second local frame, which is only accessible to that function. To apply a user-defined function to some arguments:

Bind the arguments to the names of the function's formal parameters in a new local frame. Execute the body of the function in the environment that starts with the frame

Note that the first step: bound names, is very tricky! As we have mentioned before, binding is only between values and variable names. So here what we are really doing, is not changing names for the parameters, but evaluating their values and assigning these values to new names! [see HW1 Q5]

- Name evaluation: a name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.
- 1.3.3 Example: Calling a User-Defined Function
- 1.3.4 Local Names
- One detail of a function's implementation that should not affect the function's behavior is the implementer's choice of names for the function's formal parameters. The simplest function of it is that the parameter names of a function must remain local to the body of the function.
- 1.3.5 Choosing Names
- Important principles in choosing function and parameter names:

Function names are lowercase, with words separated by underscores. Descriptive names are encouraged.

Function names typically evoke operations applied to arguments by the interpreter, or the name of the quantity that results.

Parameter names are lowercase, with words separated by underscores. Single-word names are preferred.

Parameter names should evoke the role of the parameter in the function, not just the kind of argument that is allowed.

Single letter parameter names are acceptable when their role is obvious, but avoid 1, 0 or 1 to avoid confusion with numerals.

- 1.3.6 Functions as Abstractions
- To master the use of a functional abstraction, it is often useful to consider 3 core attributes: the domain of a function is the set of arguments it can take the range of a function is the set of values it can return

the intent of a function is the relationship it computes between inputs and output (as well as any side effects it might generate)

## 1.3.7 Operators

distinguish among / (normal division, which results to a floating point, or truediv in operator), //
(rounds the result to an integer, or floordiv in operator), and % (mod)

### 1.4 Designing Functions

The qualities of good functions:
 each function should have exactly one job
 don't repeat yourself (DRY principle)
 functions should be defined generally

### 1.4.1 Documentation

- A function definition will often include documentation describing the function, called a docstring, which must be indented along with the function body. Docstrings are conventionally triple quoted. The first line describes the job of the function in one line. The following lines can describe arguments and clarify the behavior of the function.
- When you call [help] with the name of a function as an argument, you see its docstring (type [q]
  to exit), like
  - >>> help(pressure)
- Comments in Python can be attached to the end of a line following the "#" symbol

### 1.4.2 Default Argument Values

>>> def pressure(v, t, n = 6):

• In Python, we can provide default values for the arguments of a function. When calling that function, arguments with default values are optional. If they are not provided, then the default value is bound to the formal parameter name instead. Below is an example:

```
>>> pressure (1, 273.25) # pass in 2 parameters 2269
>>> pressure (1, 273.15, 6)
2269
```

### 1.5 Control

• Control statements are statements that control the flow of a program's execution based on the results of logical comparisons.

### 1.5.1 Statements

- So far, we have seen 3 statements already: def, assignment, and return.
- Rather than being evaluated, statements are executed. Each statement describes some change to the interpreter state, and executing a statement applies that change.

### 1.5.2 Compound Statements

• In general, Python code is a sequence of statements. A simple statement is a single line that doesn't end in a colon. A compound statement is so-called because it's composed of other statements. Compound statements typically span multiple lines and start with a one-line header ending in colon, which identifies the type of statement. Together, a header and an indented suite of statements is called a clause. A compound statement consists of one or more clauses.

,	neader>:									•				
	<statement></statement>													
	<statement></statement>													
<b>&lt;</b> S	eparating header>:													
	<statement></statement>													
	<statement></statement>													
· We	can catalog the sta	teme	nts w	e hav	e lear	nt:								
Ex	kpressions, return s	tatem	nent,	and a	ssignn	nent s	taten	nents	are si	mple	stater	nents.		
	def statement is a													
	can understand mul													ne first
	ement. If that stat													
	ence of statements				r eqir e			men	proce	eu io	CACCO		ie 1,65	oi me
					-n+	•							•	
	efining Functions I						٠							C::11:
	never a user-define													
	ecuted in a local e													
	function. A return													
	first return statem				and t	the va	lue of	the	returr	ı expr	ession	is, th	e reti	urned
	of the function be		pplied	d		٠								0
	Conditional Stateme													
<ul><li>A col</li></ul>	nditional statement	in Py	thon	consi	sts, of	a ser	ries of	head	ers ar	nd şui	ts: a r	equir	ed "if	" clause
an of	otional sequence of	"elif"	clau	ises, c	and, fir	nally o	an opt	ional	"else"	claus	se:			
<b>&gt;&gt;</b>	> if <expression>:</expression>													
	<suite></suite>													
<b>&gt;&gt;</b>	> elif <expression>:</expression>													
	<suite></suite>													
>>	> else:													
	<suite></suite>													•
• The	computational proc	ess of	exec	cutina	a cor	ndition	al cla	use f	ollows		•	•	•	•
	aluate th header's										•			•
	it is a true value,				to Th	on cl	in all	subse	onuan	t clau	ses in	the c	onditi	ional
		EVEC	11¢ 11	ile sui	16., 11	1611, 31	vip att	SUDS	equen	ı cıau	3e3 III	iiie c	Jonarri	Origi
statem		, , ,		, ,			ond:4:	المص	la alta	!	heal-		m t c t	
	expressions inside t													
	n value matter to c													
Pyth	on includes several	talse	value	es, inc	cluding	g 0, N	one, "	", [], d	and th	e boo	lean v	/alue	False.	All

• Python has two boolean values: True and False.

other numbers are true values.

- Python has three boolean operators: and, or, and not, the first two of which has the so-called short-circuit evaluation behavior.
- 1. [not] returns the opposite truth value of the following expression (so not will always return either True of False).
- 2. [and] evaluates expressions in order and stops evaluating (short-circuits) once it reaches the first false value, and then returns it. If all values evaluate to a true value, the last value is

0	returnea.												
3.	[or] short-circuits at	the first	true, v	value and	d retu	urņs it	. Įf d	ıll valu	ies e	valuate	to a	false	value,
	the last value is retu	rned.											
>>>	not None		0									0	
Tru	re i i												
>>>	-1 and 0 and 1												
0													0
>>>	False or 999 or 1/0												
99	9												
1.5	5.5 Iteration												
. /	A while clause structu	re is like:											
	>>> while <expressio< td=""><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></expressio<>												
	<suite></suite>												•
1.5	5.6 Testing												
	Testing a function is th	ne act of v	verify	ina that	the	functio	on's b	ehavio	or mo	atches e	xpec	tation	s. A
	test is a mechanism fo												
	of another function the												
	returned value is then												2
	meant to be general, t												
	Tests also serve as doc												
	values are suitable.			-, -,						ion, and	,	9	
	assert: when the expre	ession bein	a ass	erted ev	aluat	es to	1 tru	e valu	e. ex	cecutina	an c	ıssert	•
	statement has no effe		•										•
	execution.				,,				٠,,,,	,			•
	>>> assert f(8) == 13	3	•	•	•	•				•	•	•	•
•	>>> def f_test():		•		•		•			•	•		•
•	assert f(2) ==	1		•	•		•		•	•	•		0
	assert f(3) ==			•		•	•			•			
•	assert f(50) ==		0	•	٠		•		•			0	0
• [	Python provides a conv		thad :	for placi	na sin	nnle te	osts /	directl	v in	the doc	strin	a of a	0
	function. THE first line												•
	followed by a blank lin												0
	addition, the docstring												n the
	nteraction can be veri											)II, III	sij, ilie
	>>> def f(n):	rieu via ji	ie uoc	.1631,1110	uule.	DEIOW	is ai	Exam	pie i	Ji usuge	•	٠	0
	""" Return the	sum of th	o fir	st n alan	nants					•	•		•
	, Kejuili ine	Suin Or in	ie iii:	oi ii eteii	lieilis		٠				•	0	0
	>>> f(2)						•				•	•	•
							•	•		•		•	
•	55		•					•					
	>>> f(10)		•										6
	458										•		
0					٠	•					•	0	0
	>>> from doctest imp	port testin	nod .										

```
>>> testmod()
          TestResults(failed = 0, attempted = 2)

    When writing Python in files, rather than directly into the interpreter, tests are typically

  written in the same file or a neighboring file with the suffix "_test.py". Then all doctests can be
  run by starting Python with the doctest command line option: [python3 -m doctest (-v)
  <python_file>]
• A test that applies a single function is called a unit test.
1. To prevent the ok autograder from interpreting print statements as output: print with 'DEBUG:'
   at the front of the outputted line
2. To open n interactive terminal to investigate a failing test for question sum_digits in assignment
   lab01: [python3 ok -q sum_digits -i]
3. To look at an environment diagram to investigate a failing test for question sum_digits in
   assignment lab01: [python3 ok -q sum_digits —trace]
1.6 Higher-Order Functions

    We need to construct functions that can accept other functions as arguments, or return

  functions as values. Functions that manipulate functions are called higher-order functions.
1.6.1 Functions as Arguments
Below is an example:
     >>> def summation(n, term):
          ... ...
    >>> def identity(x):
     >>> def sum_naturals(x, identity):
1.6.2 Functions as General Methods

    With higher-order functions, we begin to see a more powerful kind of abstraction: some functions

  express general methods of computation, independent of the particular functions they call.

    We learnt about two related big ideas in computer science:

    Naming and functions allow us to abstract away a vast amount of complexity.

2. It is only by virtue of the fact that we have an extremely general evaluation procedure for the
   Python language that small components can be composed into complex processes.
• 1.6.3 Defining Functions III: Nested Definitions
• There are two negative consequences of passing functions as arguments:
1. The global frame becomes cluttered with names of small functions, which must all be unique.
2. We are constrained by particular function signatures
-> Nested function definitions address both of these problems, but require us to enrich our
environment model.
Below is an example:
     >>> def sqrt(a):
         def sqrt_update(x):
               return average(x, a/x)
         def sqrt_close(x):
```

```
return approx_eq(x * x, a)
          return improve(sqrt_update, sqrt_close)
     Note that we now place function definitions inside the body of other definitions.
• Like local assignment, local def statements only affect the local frame. These functions are only
  in scope while sart is being evaluated. Locally defined functions also have access to the name
  bindings in the scope in which they are defined. This discipline of sharing names among nested
  definitions are called lexical scoping.
• We require two extensions to our environment model to enable lexical scoping:
1. Each user-defined function has a [parent environment where it was defined.
2. When a user-defined function is called, its local frame extends its parent environment.
• Functions values each have a new annotation that we will include in environment diagrams from
  now on: a parent. The parent of a function value is the first frame of the environment in which
  that function was defined. Functions without parent annotations were defined in the global
  environment. When a user-defined function is called, the frame created has the same parent as
  that function.
1.6.4 Functions as Returned Values

    Function composition: h(x) = f(q(x))

1.6.5 Example: Newton's Method
1.6.6 Currying
• Given a function f(x, y), we can define a function q such that q(x)(y) = f(x, y). Here, q is a higher-
  order function that takes in a single argument x and returns another function that takes in a
  single argument y. This transformation is called <mark>currying</mark>. Currying is useful when we require a
  function that takes in only a single argument (like the map function).
• Inverse Currying transformation:
1. given f, return g <-currying
>>> def curry(f):
     def g(x):
          def h(y):
               return f(x, y)
          return h
     return g
2. given g, return f <-uncurrying
>>> def uncurry(g):
     def f(x, y):
          return q(x)(y)
     return f
>>> uncurry(curry(f))
f
1.6.7 Lambda Expressions
• In Python, we can create function values on the fly using lambda expressions, which evaluate to
  unnamed functions. A lambda expression evaluates to a function that has a single return
  expression as its body. Assignment and control statements are not allowed. Below is an example:
```

>>> def compose(f, g):

return lambda x: f(g(x))

We can understand the s	tructure	of a la	ambda e	xpression	by co	nstruct	ing a	corres	ponding	g sent	ence:			
lambda	<b>x</b>	: .	•	f(g(x))										
A function that take	s x, and	l return	ns f(g	g(x))										
• The result of a lambda	express	ion is c	alled a	lambda f	unction	. It has	no i	ntrinsic	name	, but				
otherwise it behaves li	ke any o	ther fu	nction.											
>>> s = lambda x; x* x														
>>> S														
<function <lambda=""> at 0x</function>	f3f490>													
>>> s(12)														
144														
>>> (lambda x: x*x)(12)														
.144														
• The main difference be	tween la	ımbda e	expressi	ons and d	lef sta	tements	is th	at, onl	y the	def				
statements give the fu	nction in	trinsiç	names.	All lambo	la expr	essions	share	the n	ame [G	reek				
statements give the function intrinsic names. All lambda expressions share the name [Greek letter(x)]  • There is one key point about lambda expressions: they cannot recursively call themselves.														
• There is one key point	about la	mbda e	xpression	ons: they	cannot	recurs	ively	call th	emselv	es.				
>>> $h = lambda x: f(h(x))$	#wrong													
>>> h = (lambda g:lambda	x: f(g(x)	) is )(h)	#corre	ect .										
1.6.8 Abstractions and Fi	rst-Class	Functi	ions											
• Programming languages	impose	restrict	tions on	the way	s in wh	ich com	puta	ional e	lemen	ts car	be			
manipulated. Elements	with the	fewes	t reștric	ctions are	said t	o be ho	ive fi	rsț-cla	ss stat	us. S	ome			
of the "rights and privi	ileges" of	f first-	class el	ements a	re:									
1 There were be becaut to	o namas													
1. They may be bound t	o mames.		0						0		0			
2. They may be passed			function	ons.										
	as argum	ents to				•	•				0 0			
2. They may be passed	as argum d as the	ents to results	of fund		0	0		•			0 0			
<ol> <li>They may be passed</li> <li>They may be returned</li> </ol>	as argum d as the l in data	ents to results structu	of fund ures.	ctions.	be ma	nipulate	ed as	values.	They	are	0 0			
<ol> <li>They may be passed</li> <li>They may be returned</li> <li>They may be included</li> </ol>	as argum d as the l in data n nature,	ents to results structu are fu	of fund ures, nctions	ctions. that can							nction			
<ol> <li>They may be passed of</li> <li>They may be returned</li> <li>They may be included</li> <li>First-class functions, in</li> </ol>	as argum d as the l in data n nature,	ents to results structu are fu	of fund ures, nctions	ctions. that can							nction			
<ol> <li>They may be passed</li> <li>They may be returned</li> <li>They may be included</li> <li>First-class functions, in opposite to higher-order</li> </ol>	as argum d as the l in data n nature, er functio	ents to results structu are fu	of fund ures, nctions	ctions. that can							nction			
<ul> <li>2. They may be passed of</li> <li>3. They may be returned</li> <li>4. They may be included</li> <li>First-class functions, in opposite to higher-order</li> <li>as a return value</li> </ul>	as argum d as the I in data n nature, er functions	ents to results structu are fu ons, whi	of fund ures, nctions ich take	ctions. that can function	ns as an	rgumen:	t yalu	es, or	return	a fur	nction			
<ul> <li>2. They may be passed of</li> <li>3. They may be returned</li> <li>4. They may be included</li> <li>• First-class functions, in opposite to higher-order</li> <li>as a return value</li> <li>1.6.9 Function Decorators</li> </ul>	as argum d as the l in data n nature, er functions s l syntax	ents to results structu are fui ons, whi	of fundares. nctions ich take	that can function r-order f	unction	rgumen is as pa	t yalu	es, or	return	a fur	nction			
<ol> <li>They may be passed of</li> <li>They may be returned</li> <li>They may be included</li> <li>First-class functions, in opposite to higher-order as a return value</li> <li>1.6.9 Function Decorators</li> <li>Python provides special</li> </ol>	as argum d as the l in data n nature, er functions s l syntax	ents to results structu are fui ons, whi	of fundares. nctions ich take	that can function r-order f	unction	rgumen is as pa	t yalu	es, or	return	a fur	nction			
<ol> <li>They may be passed of</li> <li>They may be returned</li> <li>They may be included</li> <li>First-class functions, in opposite to higher-order as a return value</li> <li>1.6.9 Function Decorators</li> <li>Python provides special statement, called a decorate</li> </ol>	as argum d as the l in data n nature, er functions s l syntax	ents to results structu are fui ons, whi	of fundares. nctions ich take	that can function r-order f	unction	rgumen is as pa	t yalu	es, or	return	a fur	nction			
<ol> <li>They may be passed of the statement, called a despectation.</li> <li>They may be included to the first-class functions, in opposite to higher-order as a return value.</li> <li>Python provides special statement, called a despectation.</li> </ol>	as argum d as the l in data n nature, er function s l syntax ecorator	ents to results structu are fui ons, whi	of fundares. nctions ich take	that can function r-order f	unction	rgumen is as pa	t yalu	es, or	return	a fur	nction			
<ol> <li>They may be passed of the statement, called a def wrapped(x):</li> </ol>	as argum d as the l in data n nature, er function s l syntax ecorator	ents to results structu are fui ons, whi	of fundares. nctions ich take	that can function r-order f	unction	rgumen is as pa	t yalu	es, or	return	a fur	nction			
<ol> <li>They may be passed of the passe</li></ol>	as argum d as the l in data n nature, er function s l syntax ecorator	ents to results structu are fui ons, whi	of fundares. nctions ich take	that can function r-order f	unction	rgumen is as pa	t yalu	es, or	return	a fur	nction			
<ol> <li>They may be passed of the passe</li></ol>	as argum d as the l in data n nature, er function s l syntax ecorator	ents to results structu are fui ons, whi	of fundares. nctions ich take	that can function r-order f	unction	rgumen is as pa	t yalu	es, or	return	a fur	action			
<ol> <li>They may be passed of the passe</li></ol>	as argum d as the l in data n nature, er function s l syntax ecorator	ents to results structu are fui ons, whi	of fundares. nctions ich take	that can function r-order f	unction	rgumen is as pa	t yalu	es, or	return	a fur	nction			
<ol> <li>They may be passed of the statement of the s</li></ol>	as argum d as the l in data n nature, er function s l syntax ecorator	ents to results structu are fui ons, whi	of fundares. nctions ich take	that can function r-order f	unction	rgumen is as pa	t yalu	es, or	return	a fur	nction			
<ol> <li>They may be passed of the statement of the s</li></ol>	as argum d as the l in data n nature, er function s l syntax ecorator	ents to results structu are fui ons, whi	of fundares. nctions ich take	that can function r-order f	unction	rgumen is as pa	t yalu	es, or	return	a fur	nction			
<ol> <li>They may be passed of the statement of the s</li></ol>	as argumd as the lin data n nature, er functions l syntax ecorator.	ents to results structu are fui ons, whi to appl Below	of fund ures. nctions ich take y highe is the n	that can function r-order f	unction	rgumen is as pa	t yalu	es, or	return	a fur	nction			
<ol> <li>They may be passed of the statement of the s</li></ol>	as argumd as the lin data n nature, er functions l syntax ecorator.	ents to results structu are fui ons, whi to appl Below	of fund ures. nctions ich take y highe is the n	that can function r-order f	unction	rgumen is as pa	t yalu	es, or	return	a fur	action			
<ol> <li>They may be passed of the statement of the s</li></ol>	as argumd as the lin data n nature, er functions l syntax ecorator.	ents to results structu are fui ons, whi to appl Below	of fund ures. nctions ich take y highe is the n	that can function r-order f	unction	rgumen is as pa	t yalu	es, or	return	a fur	nction			

>>> def triple(x): return 3\*x

>>> triple = trace(triple)

• The decorator symbol "@" may also be followed by a call expression. The expression following @ is evaluated first (just as the name trace was evaluated first), the def statement second, and finally the result of evaluating the decorator expression is applied to the newly defined function, and the result if bound to the name in the def statement.

### 1.7 Recursive Functions

- A function is called recursive if the body of the function calls the function itself.
- 1.7.1 The Anatomy of Recursive Functions
- A common pattern can be found in the body of many recursive functions. the body begins with a
  base case, a conditional statement that defines the behavior of the function for the inputs that
  are simplest to process. Some recursive functions will have multiple base cases. They are then
  followed by one or more recursive calls, which always have a certain character: they simplify the
  original problem incrementally.
- Treating a recursive call as a functional abstraction has been called a recursive leap of faith.

  We define a function in terms of itself, but simply trust that the simpler cases will work

  correctly when verifying the correctness of the function. Verifying the correctness of a recursive
  function is a form of proof by induction.

### 1.7.2 Mutual Recursion

- When a recursive procedure is divided among two functions that call each other, the functions are said to be mutually cursive.
- 1.7.3 Printing in Recursive Functions
- 1.7.4 Tree Recursions
- Another common pattern of computation is called tree recursion, in which a function calls itself more than once.
- A function with multiple recursive calls is said to be tree recursive, because each of call branches into multiple smaller calls, each of which branches into yet smaller calls, just as the branch of a tree become smaller but more numerous as they extend from the trunk.
- 1.7.5 Example: Partitions

[2] Building Abstract	ions w	ith Dat	ta											
2.1 Introduction														0 (
2.1.1 Native data Ty	pes													
• Every value in Pyt	họn họ	as a cla	iss th	at de	etermi	nes w	hat t	ype of	valu	e it is.	The	built-	in [ty	pe]
function allows us	to ins	pect th	ne cla	ss of	any v	alue.								
>>> type(2)														
<class 'int'=""></class>													•	
• Native data types	have	the fol	lowing	g pro	pertie	s:								
1. There are expres	ssions	that ev	aluat	te to	value	s of n	ative	types	, calle	ed lite	rals.			
2. There are built-i	n func	tions a	nd op	erate	ors to	mani	pulate	value	s of	native	types	5,	0	0 (
<ul> <li>Python includes the numbers).</li> </ul>	ree nu	umeric	types	; int	(integ	ers),	float (	(real r	numbe	ers) an	d com	plex	(comp	lex
• The difference bet	tween	int and	I floa	t is v	ery in	nporto	ant: in	t obje	ects r	eprese	ent in	teaer	s exac	:tlv:
float objects can														
represented exact				g-	•	٠,٥٠								•
• There are also non		eric tvo	es of	data	ı. suck	as t	he boo	ol clas	s for	values	· s True	and	False.	0 0
, more are alocation	, ,, anno	-1,0 17F			.,			o, orac		, at a c		, and	, 4,00.	
2.2 Data Abstraction	n	•	•	•	•	•			•	•	•		•	
• The general techn		f isolat	ina tl	ne na	rts of	a pr	oaram	that	deal	with h	ow do	· ita ar	ė	0
represented from														
methodology called														an.
maintain, and modi		uporre		. Dui	u ubo	.,	on ma	iii Pi	yg. u.	no ma	on ou	oici i	0,000	9'''
2.2.1 Example: Ratio		ımhers	•	•	•		•	•	•	•	•	•	•	
2.2.2 Pairs	iiqi ive	AI (IDCI S	•	•	•				•	•	•		0	
• List:	•		•	•	•		•	•	•		•			
>>> pair = [10, 20]		•	•	•	•	•		•	•			•		
>>> x, y = pair	٠	•	•	•	•		•	•	•	•	•	•	0	0
>>> X	•		•	•				•	•	•	•	•		
10	•		•	•		٠			•	•			0	
>>> pair[0]		•	•	•		٠			•		•	•		0
10		•	•	•			•	•	•	•			•	•
>>> from operator im	nort c	ratitam	•	•	0	٠		•	•	•		•	0	0 (
>>> getitem(pair, 0)	ipor i g	gennem				•					•		•	0
	٠	•		•					•		•		•	
10		•	•	0	0	٠		•					0	0
2.2.3 Abstraction Ba								, T.,	مامام	41	· c	, 		
• Function can be di														
abstraction barrie														
level of abstraction											r,T Of	The p	rograr	n that
can use higher lev			nstea	d use	es a tu	ıņctio	n in a	lower	leve	l.,				0 0
2.2.4 The Properties	of da	та						•			•			
2.3 Sequences			•										٠	0 (
• A sequence is an o				of val	lues.	here	are n	nany d	liftere	eņt kir	ids of	sequ	ences,	but
they all share com	mon h	ehavior	cc.											

- 1. A sequence has a finite length. An empty sequence has length 0.
- A sequence has an element corresponding to any non-negative integer index less than its length, starting from 0 for the first element.

```
2.3.1 Lists
```

```
>>> digits = [1, 8, 2, 8]
```

>>> len(digits)

4

>>> [2, 7] + digits \* 2

[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]

>>> pairs = [[10, 20], [30, 40]]

>>> pairs[1, 0]

30

## 2.3.2 Sequence Iteration

- In many cases, we would like to iterate over the elements of a sequence and perform some computation for each element in turn. This pattern is so common that Python has an additional control statement to process sequential data: the for loop.
- A for statement consists of a single clause with the form:

for <name> in <expression>:

<suite>

And a for statement is executed by the following procedure:

- 1. Evaluate the header (expression), which must yield an iterable value.
- 2. For each element value in that iterable value, in order:
  - 1. Bind <name> to that value in the current frame
  - 2. Execute the <suite>
- A common pattern in programs is to have a sequence of elements that are themselves sequences, but all of a fixed length. A for statement may include multiple names in its header to "unpack" each element sequence into its respective elements. Such pattern of binding multiple names to multiple values in a fixed-length sequence is called sequence unpacking.

```
>>> pairs = [[1, 2], [2, 2], [2, 3], [3, 4]]
```

>>> same = False

>>> for x, y in pairs:

if x == y:

same = True

>>> same

True

- A range is another built-in type of sequence in Python, which represents a range of integers. range takes in two integer arguments: the first number and one beyond the last number in the desired range. If only one argument is given, it is interpreted as one beyond the last value for a range that starts at 0.
- A common convention is to use a single underscore character for the name in the for header if the name is unused in the suite. This underscore is just another name in the environment as far as the interpreter is concerned, but has a conventional meaning among programmers that indicates the name will not appear in any future expressions.
- 2.3.3 Sequence Processing

• Many sequence element in a s																h.
comprehension																
>>> odds = [1, 3,						.,							•		•	
>>> [x+1 for x ir				•		•	٠	•	•		•	•		٠	•	
[2, 4, 6, 8, 10]	, odd	•	•		•	•	•	•	•	•	•	•	•		•	
Notice the abov	e for	kevw	ord is	conto	ined i	within	SOUR	re hre	ickets	the	refore	it is	not r	oart o	f a foi	•
statement, but																
bound to each e															^	
• List comprehe																
>>> [x for x in o					Selec	ı a su	DSEI	oj vai	ues III	ıqı se	aijaiy	SOME	Condi	TIÇII.	٠	
[1, 5]	uus II	25 /6	, , ==	. ()			•	•		•		•				
	m of .	a lict	comp	coboni	ion ic	•		•	•	•	•		•	0	٠	
The general for							.cian.	:£ ,£;;	i Itan a		.cion. 1					
[ <map expression<="" td=""><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>:</td><td></td><td></td><td></td></map>													:			
To evaluate a li																,n
iterable value.																
expression is ev							iue, t	ne ma	ip exp	ressi	on is e	valua	теа. 1	ine va	iues o	Γ,
the map express																
• A third commo												n a s	equen	ce int	o a	
single value.																
• The common p		ns we	have	obse	rved 1	n sequ	ience	proce	ssing	can b	e exp	resse	dusin	g high	er-	
order function		•														
2.3.4 Sequence						•										
• A value can b															t in]	
that evaluate	s to 1	rue o	r. Fals	se dep	endin	g on w	heth	er an	eleme	nt ap	pears	in a	seque	nçe.		
>>> digits						0		0						0		
[1, 8, 2, 8]																
>>> 2 in digits						0	•	0						0	0	
True																
• Sequence con	tain s	maller	· sequ	iences	withi	in the	m. A	slice o	of a se	quen	ice is	any c	ontigu	ious sp	an of	
, the original se																
integer indica															nding	
index. In Pyth	ion, s	equen	ce slic	cing is	expr	essed	simila	irly to	elem	ent s	electi	on, us	sing so	quare		
brackets, A c	olon s	epara	tes th	he sta	irting	and e	nding	indice	es. Any	, bou	ind the	at is	omitte	ed is a	ssume	d
to be an extr	eme v	alue:	0 for	the s	tartir	ng inde	ex, an	d the	lengt	h of	the se	quen	ce for	the e	nding	
index.																
>>> digits[0:2]																
[1, 8]																
>>> digits[1:]																
[8, 2, 8]																
2.3.5 Strings																

- The native data type for text in Python is called a string, and corresponds to the constructor str.
- String literals can express arbitrary text, surrounded by either single or double quotation marks (If you use double quotes, then you can use single quotes inside of it. Otherwise you cannot.).

- The elements of a string are themselves strings that have only a single character. A character is
  any single letter of the alphabet, punctuation mark, or other symbol. Unlike many other
  programming languages, Python does not have a separate character type.
- Like lists, strings can also be combined via addition and multiplication.

>>> 'Berkeley' + ', CA'

'Berkeley, CA'

>>> 'Shabu' \* 2

'Shabu Shabu'

• The membership operator [in] applies to strings, but has an entirely different behavior than when it is applied to sequences. It matches substrings rather than elements.

>>> 'here' in 'Where's she'

#### True

- Strings are not limited to a single line. Triple quotes delimit string literals that span multiple
  lines. '\n' is a single element that represents a new line, and is considered a single character for
  the purpose of length and element selection.
- A string can be created from any object in Python by calling the str constructor function with an object value as its argument

### 2.3.6 Trees

- Our ability to use lists as the elements of other lists provides a new means of combination in our programming language. The ability is called a closure property of a data type. In general, a method for combining data values has closure property if the result of combination can itself be combined using the same method. Closure is the key to power in any means of combination because it permits hierarchical structures.
- Nesting lists within lists can introduce complexity. The tree is a fundamental data abstraction that imposes regularity on how hierarchical values are structured and manipulated. A tree has a root label and a sequence of branches. Each branch of a tree is a tree. A tree with no branches is called a leaf. Any tree contained with a tree is called a sub-tree of that tree. The root of each sub-tree is called a node in that tree. A tree is well-formed only if it has a root label and all branches are also trees.

## 2.3.7 Linked Lists

- A common representation of a sequence constructed from nested pairs is called a linked list.
- Linked lists have recursive structure: the rest of a linked list or 'empty'.

### 2.4 Mutable Data

We need strategies to help us structure large systems to be modular, meaning that they divide
naturally into coherent parts that can be separately developed and maintained. One powerful
technique for creating modular programs is to incorporate data that may change state over time.
In this way, a single data object can represent something that evolves independently of the rest
of the program. Adding state to data is a central ingredient of a paradigm called object-oriented
programming.

### 2.4.1 The Object Metaphor

 Objects combine data values with behavior. Objects are both information and processes, bundled together to represent the properties, interactions, and behaviors of complex things.

>>> from datetime import date

The name [date] is bound to a class. As we have seen, a class represents a kind of value. Individual dates are called instances of that class. Instances are constructed by calling the class on arguments that characterize the instance.

• Objects have attributes, which are named values that are part of the object. in Python, like many other programming languages, we use dot notation to designate an attribute of an object. <expression>.<name>

The <expression> evaluates to an object, and <name> is the name of an attribute for that object. These attributes are not available in the general environment.

- Objects also have methods, which are function-valued attributes. By implementation, methods are
  functions that compute their values from both their arguments and their object. The expression
  for attribute also applies to methods.
- All values in Python are objects.
- 2.4.2 Sequence Objects
- Instances of primitive built-in values such as numbers are immutable. The values themselves cannot change over the course of program execution. Lists on the other hand are mutable.
- Mutable objects are used to represent values that change over time. An object may have changing properties due to muting operations. Most changes are performed by invoking methods on list objects.
- Here are many list modification operations:
- >>> chinese = ['coin', 'string', 'myriad']
- >>> suits = chinese
- >>> suits.pop()

'myriad' #remove and return the final element

- >>> suits.remove('string') #remove the first element that equals the argument
- >>> suits.append('cup') #add an element to the end
- >>> suits.extend(['sword', 'club']) #add all elements of a sequence to the end
- >>> suits[2] = 'spade' #replace an element
- >>> suits[0:2] = ['heart', 'diamond'] #replace a slice
- >>> suits

['heart', 'diamond', 'spade', 'club']

Note all of these mutation operations change the value of the list; they do not create new list objects. Since the object bound to the name chinese has also changed, because it is the same list object that was bound to suits! This behavior is different from what we have learned from primitive data types. With mutable data, methods called on one name can affect another name at the same time.

• Lists can be copied using the [list] constructor function. Changes to one list do not affect another, unless they share structure.

>>> new = list(suits)

Note that we can use slicing to shallow copy lists, but actually the mechanism is: for lists or something that we must store in memory (not other prmitive data types like ints), we copied the references to them instead of the items themselves. Therefore, I can say that both elements [is] the same thing, but modifying one element means changing that reference without change the item, and the other will not be affected.

>>> suits.insert(2, 'Joker') #insert an element at index 2, shifting the rest

<ul> <li>Because two lists n</li> </ul>	nay ho	ave th	e san	ne con	tents	but ir	n, fact	be di	ffere	nt list	s, we	requi	re a n	neans
, to test whether tw	o obj	ects a	re th	e sam	e. Py	thon i	nclude	s two	comp	arison	oper	ators,	calle	d [is]
and [is not], that to	eșt wh	nether	two	expre	ssions	in fa	ct evo	luate	to th	ne ider	ntical	objec	t. Tw	Q
objects are identic	al if t	hey a	re eq	ual in	their	curre	nt val	ue, ai	nd any	y chan	ge to	on w	ill alw	ays
be reflected in the	other	r. <mark>Ide</mark> r	ntity	is a st	ronge	er con	dition	than	equal	ity.				
>>> suits is ['heart, 'd	iamon	ď, 'sp	ade',	'club']	#che	eck fo	r iden	tity						
False														
>>> suits == ['heart, 'd	diamor	nd', 'sp	oade',	'club'	] #ch	eck f	or equ	ality						
True														
• A list comprehension	n alw	ays cr	eates	s a nev	w list									
• It is very dangerou							the de	fault	value	for a	func	tion a	raume	ent
(list lists). It is bed														
arguments are only														
Therefore, every ti														•
mutable).		o ran		io cuit	Çu,	o acro	411 10	i,uc v	,,, DC	çııanış	,cu (c			•
• majable);	•		•	•	0	٠	•	•	•	•	•	•	0	٠
• A tuple, an instance	a af +	ha hui	il+ in	tunla	tuna	is an	immu	table.		nco T	Tunlos	ara	· ·roate	.d
using a tuple litera														
										Paren	inese	s are	opnon	iại bu
used commonly in p	гастіс	e. Any	y opje	ECTS C	an be	place	a into	Tuple	2\$.					
>>> 1, 2 + 3						٠			•					
(1, 5)							•						•	
There are also empty	and d	oņe-el	emen	t tuple	es wh	içh ha	s spe	cial sy	ntax.	•				
>>> ().													0	
.0														
>>> tuple()														
.0					0							•		
>>> (10, )														
(10, )														
Here are some metho	ds of	tuples	whic	ch are	share	ed bet	ween	it and	lists	: in &	not i	n; mul	and	add;
len; []; slice														
>>> code = (1, 1, 2, 2,	3, 4,	3, 4)												
>>> code.count(2)														
2														
>>> code.index(3)													0	
4														
But note that the me	thods	for m	nanipu	lating	the	conter	its of	a list	are	not av	ailabl	e for	tuples	5,
because they are imm														
within a tuple.					•	J								
>>> nested = (10, 20,	[30. 4	.01)	•	•	•	•	•		•		•		•	
>>> nested[2].pop()	[50, 1	41)	•		٠	•	•	•	•		•	•	•	•
Note that you can ch	ange	the el	emen	te inci	de of	4 mu		alama	nt hi	It vou	cann	ot cha	Inae t	he
whole element itself,								e cine	יוןי, טכ	ai you	Cum	OI CIIC	ilige i	ije
>>> lst = [1, 2, [0]]	(01 11	is ha	1,1 01				Apre.			•		•		0
	•			•			•						•	
>>> lst[2] = [1]						•								•

### Error

Tuples are often used implicitly in multiple assignment. An assignment of two values to two names creates a two-element tuple and then unpacks it.

#### 2.4.3 Dictionaries

- Dictionaries are Python's built-in data type for storing and manipulating correspondence relationships. A dictionary contains key-value pairs, where both the keys and values are objects.
   The purpose of a dictionary is to provide an abstraction for storing and retrieving values that are indexed not by consecutive integers, but by descriptive keys.
- Look up values:

```
>>> numerals = ['i': 1.0, 'v': 5, 'x':10]
```

>>> numerals['x']

10

• A dictionary can have at most one value for each key. Adding new key-value pairs and changing the existing value for a key can both be achieved with assignment statements:

```
>>> numerals['i'] = 1
```

>>> numerals['l'] = 50

Note that the newly added key-values pair is not added to the end of the dictionary, because dictionaries are unordered collections of key-value pairs. When we print a dictionary, the keys are values are rendered in some order, but as users of the language we cannot predict what the order will be. The order may change when running a program multiple times.

- The methods keys, values and items all return iterable values.
- A list of key-value pairs can be converted into a dictionary by calling the [dict] constructor.
- Dictionaries have some restrictions:
- 1. A key of a dictionary cannot be or contain mutable value.
- 2. There can be at most one value for a given key.

According to the first restriction, tuples are often used as keys in dictionaries instead of lists. On the other hand, a list not allowed be used as keys for dictionaries (unhashable list). Tuples can contain anything, including lists; but those containing a list as an element cannot be used as keys.

A useful method implemented by dictionaries is [get], which returns either the value for a key, if
the key is present or a default value. The arguments to get are the key and the default value.

```
>>> numerals.get(`a', 0)
```

0

>>> numerals.get(`v', 5)

5

• We can also remove one key-value pair using key

```
>>> numerals.pop('v')
```

5

 Dictionaries have a comprehension syntax analogous to those of lists. A key expression and a vaue expression are separated by a colon. Evaluating a dictionary comprehension creates a new dictionary object.

```
>>> [x: x * x fr x in range(3, 6)]
```

```
[3:9;4:16;5:25]
```

2.4.4 Local State

Lists and dictionaries have local state: they are changing values that have some particular

contents at any point in the execution of a program. The word "state" implies an evolving process in which that state may change.

• Functions can also have local state.

nonlocal balance

The nonlocal statement declares that whenever we change the binding of the name balance, the binding is changed in the first frame in which balance is already bound. If balance has not previously been bound to a value, then the nonlocal statement will give an error.

- Ever since we first encountered nested def statements, we have observed that a locally defined function can look up names outside of its local frames. No nonlocal statement is required to access a non-local name. By contrast, only after a nonlocal statement can a function change the binding of names in these frames.
- Note that in our example, balance is bound to an immutable value (an integer). If we bound a temporary variable to a mutable value, like a list, we can refer to or modify the value of elements inside of it in the local frame, without defining non-local.
- 2.4.5 The Benefits of Non-Local Assignment
- Non-local assignment is an important step on our path to viewing a program as a collection of independent and autonomous objects, which interact with each other but each manage their own internal state. In particular, non-local assignment has given us the ability to maintain some state that is local to a function, but evolves over successive calls to that function.
- 2.4.6 The Cost of Non-Local Assignment
- When two names are both bound to a function, it does matter whether they are bound to the same function, or different instances of that function. By introducing non-pure functions that change the non-local environment, we have changed the nature of expressions. An expression that contains only pure function calls is referentially transparent: its value does not change if we substitute one of its subexpression with the value of that subexpression. Re-binding operations violate the conditions of referential transparency because they do more than return a value.
- 2.4.7 Implementing Lists and Dictionaries
- None is None, but [] is not [] (they are equal, but not identical).
- We design our mutable linked list function as a dispatch function, and its arguments are first a message, followed by additional arguments to parameterize that method. This message is a string naming what the function should do. Dispatch functions are effectively many functions in one: the message determines the behavior of the function, and the additional arguments are used in that behavior.
- The approach, which encapsulates the logic for all operations on a data value within one function that responds to different messages, is a discipline called message passing. A program that uses message passing defines dispatch functions, each of which may have local state, and organizes computation by passing "messages" as the first argument to those functions. The messages are strings that correspond to particular behaviors.
- 2.4.8 Dispatch Dictionaries
- 2.4.9 Propagating Constraints
- Declarative programming: in which a programmer declares the structure of a program to be solved, but abstracts away the details of exactly how the solution to the problem is computed.
- Computer programs are traditionally organized as one-directional computations, which perform operations on pre-specified arguments to produce desired outputs. On the other hand, we often

want to model systems in terms of relations among quantities. Such a model is not one-directional.

• We can use a message passing system to coordinate constraints and connectors. Constraints are dictionaries that do not hold local states themselves. Their responses to messages are non-pure functions that change the connectors that they constrain. Connectors are dictionaries that hold a current value and respond to messages that manipulate that value. Constraints will not change the value of connectors directly, but will do so by sending messages, so that the connector can notify other constraints in response to the change. In this way, a connector represents a number, but also encapsulates connector behavior.

# 2.5 Object-Oriented Programming

## 2.5.1 Objects and Classes

- A class serves as a template for all objects whose type is that class. Every object is an instance of some particular class. The objects we have used so far all have built-in classes, but new user-defined classes can be created as well. A class definition specifies the attributes and methods shared among objects of that class.
- The act of creating a new object instance is known as instantiating the class. The syntax in Python for instantiating a class is identical to the syntax of calling a function.

>>> a = Account('Kirk')

- An attribute o an object is a name-value pair associated with the object, which is accessible via
   dot notation. The attribute specific to a particular object, as opposed to all objects of a class,
   are called instance attributes.
- Functions that operate on the object or perform object-specific computations are called methods. 2.5.2 Defining Classes
- User-defined classes are created by class statements, which consist of a single clause. A class statement defines the class name, then includes a suite of statement to define the attributes of the class.

class <name>:

<suite>

An example:

class Account:

```
def __init__(self, account_holder): #constructor
    self.balance = 0
    self.holder = account_holder
```

The \_\_init\_\_ method has two formal parameters. The first one, self, is bound to the newly created object. The second one, is bound to the argument passed into the class when it is called to be instantiated.

• Each new object instance has its own attribute, the value of which is independent of other objects of the same class. To enforce this separation, every object that is an instance of a user-defined class has a unique identity. Object identity is compared using is and is not operators. As usual, binding an object to a new name using assignment does not create a new object.

```
>>> c = a
>>> c is a
```

True

```
New objects that have user-defined classes are only created when a class is instantiated with call
expression syntax.
2.5.3 Message Passing and Dot Expressions
• The built-in function [getattr] returns an attribute for an object by name.
>>> getattr (spock_account, 'balance')
10
(getattr and dot expressions look up a name in the same way)
• The built-in function [hasattr] checks whether an object has a named attribute.
>>> hasattr (spock_account, 'balance')

    The difference between functions and bound methods:

1. Functions, which we have been creating since the beginning of the course.
2. Bound methods, which couple together a function and the object on which that method will be
   invoked. (Object + Function = Bound Method)
>>> type(Account.deposit)
<class 'function'>
>>> type(spock_account.deposit)
<class 'method'>
The first is a standard two-argument function with parameters self and amount. The second is a
one-argument method, where the name self will be bound to the object when the method is called.
We can call deposit in two ways:
>>> Account.deposit(sa, 100) #the deposit function takes two arguments
>>> sa.deposit(100) #the deposit function take one argument
• Class names are conventionally written using the CapWords convention
2.5.4 Class Attributes

    Some attribute values are shared across all objects of a given class. Such attributes are

  associated with the class itself, rather than any individual instance of the class. Class attributes
  are created by assignment statements in the suite of a class statement, outside of any method
 definition (they may also be called class variables or static variables.)
class Account:
    interest = 0.02
    def __init__(self, account_holder): #constructor
         self.balance = 0
         self.holder = account_holder
A single assignment statement to a class attribute changes the value of the attribute for all
instances of the class. However, it is not vice versa.
>>> Account.interest = 0.04
>>> account.interest
>>> account.interest = 0.08
>>> Account.interest
0.04
• Dot expression consists of an expression, a dot, and a name:
<expression>.<name>
```

<ol> <li>evaluate the expressions to the left of the dot, which yields the object of the dot exp</li> <li>evaluate the expressions to the left of the dot, which yields the object of the dot exp</li> <li>example is matched against the instance attributes of the object; if an attribute with the name exists, its value is returned</li> <li>if enames does not appear among instance attributes, then enames is looked up in the classical which yields a class attribute value.</li> <li>that value is returned unless it is a function, in which case a bound method is returned Objects receive messages via dot notation.</li> <li>tom_account.deposite(10)</li> </ol>	ass,
name exists, its value is returned  3. if <name> does not appear among instance attributes, then <name> is looked up in the cl which yields a class attribute value.  4. that value is returned unless it is a function, in which case a bound method is returned Objects receive messages via dot notation.</name></name>	ass,
<ol> <li>if <name> does not appear among instance attributes, then <name> is looked up in the cl which yields a class attribute value.</name></name></li> <li>that value is returned unless it is a function, in which case a bound method is returned Objects receive messages via dot notation.</li> </ol>	
which yields a class attribute value.  4. that value is returned unless it is a function, in which case a bound method is returned Objects receive messages via dot notation.	
4. that value is returned unless it is a function, in which case a bound method is returned Objects receive messages via dot notation.	instead.
Objects receive messages via dot notation.	instead.
tom_account.deposite(10)	
[Dot expression]	
[Call expression]	
2.5.5 Inheritance	
• Two classes may have similar attributes, but one represents a special case of the other.	B is a
specialization of A. In OOP terminology, the generic class (A) will serve as the base class	of B.
The terms parent class and superclass are also used for the base class, while child class	is also
used for the subclass.	
• A subclass inherits the attributes of its base class, but may override certain attributes, i	ncluding
certain methods. With inheritance, we only specify what is different between the subclas	s and
the base class. Anything that we leave unspecified in the subclass is automatically assum	ed to
behave just as it would for the base class.	
• is-a and has-a relationship	
2.5.6 Using Inheritance	
• The implementation of B:	
>>> class Account:	
>>> class CheckingAccount(Account):	
We can define the procedure to look up a name in a class recursively:	
1. If it names an attribute in the class, return the attribute value.	
2. Otherwise, look up the name in the base class, if there is one.	
• Attributes that have been overridden are still accessible via class objects.	
>>> class CheckingAccount(Account):	
withdraw_charge = 1	
interest = 0.01	
def withdraw(self, amount):	
return Account.withdraw(self, amount + self.withdraw_charge)	
2.5.7 Multiple Inheritance	
<ul> <li>Python supports the concept of a subclass inheriting attributes from multiple base cases,</li> </ul>	1
language feature called multiple inheritance.	<b>~</b> .
>>> class SavingsAccount(Account):	
/// ciass savings/recognitives	0 0
>>> class specialAccount(CheckingAccount, SavingsAccount):	
>>> class special account checking account, sayings accounts.	0 0
• For a diamond shape inheritance relationship, Python resolves names from left to right, th	

upwards. This is vital when it comes to find names for ambiguous references.

## 2.5.8 The Role of Objects

### 2.9 Recursive Objects

• Objects can have other objects as attribute values. When an object of some class has an attribute value of that same class, it is a recursive object.

### 2.9.1 Linked List Class

- A linked list is composed of a first element and the rest of the list. The rest of a linked list is itself a linked list a recursive definition. The empty list is a special kind of linked list that has no first element or rest. A linked list is a sequence: it has a finite length and supports element selection by index.
- The link class has the closure property. A link can contain a link as its first element.
- Recursive functions are particularly well-suited to manipulate linked lists.
- 2.9.2 Tree Class

### 2.9.3 Sets

• In sets, duplicate elements are removed upon construction. Sets are unordered collection.

Common built-in functions for sets:

```
>>> 3 in s #membership tests
```

True

```
>>> len(s) #length computation 4
>>> s.union({1, 5}) #set union {1, 2 3, 4, 5}
>>> s.intersection({6, 5, 4, 3}) {3, 4}
```

Sets are mutable, and can be changed one element at a time using add, remove, discard, and pop.

# 2.7 Object Abstraction

• A central concept in object abstraction is a generic function, which is a function that can accept values of multiple different types. We will consider three different techniques for implementing generic functions: shared interfaces, type dispatching, and type coercion.

## 2.7.1 String Conversion

- To represent data effectively, an object should behave like the kind of data it is meant to represent, including producing a string representation of itself.
- Python stipulates that all objects should produce two different string representations: one that is human-interpretable text and one that is a Python-interpretable expression. The constructor function for strings, [str], returns a human-readable string. Where possible, the [repr] function returns a Python expression that evaluates to an equal object. The result of calling repr on the value of an expression is what Python prints in an interactive session. Oftentimes str and repr are the same functions.

.1
>>. print(repr(a))
1 #the two expressions are the same
• We would like them to be generic or polymorphic functions, which can be applied to many
different forms of data. The object system provides an elegant solution in this case: the repr
function always invokes a method calledrepr on its argument. (the same, str ->str)
• By implementing this same method in user-defined classes, we can extend the applicability of rep
to any class we create in the future. This example highlights another benefit of dot expressions
in general: they provide a mechanism for extending the domain of existing functions to new
object types.
• These polymorphic functions are examples of a more general principle: certain functions should
apply to multiple data types. Moreover, one way to create such a function is to use a shared
attribute name with a different definition in each class.
2.7.2 Special Methods
• In Python, certain special names are invoked by the Python interpreter in special circumstances.
For instance, theinit method of a class is automatically invoked whenever an object is
constructed. Thestr method is invoked automatically when printing, andrepr is
invoked in an interactive session to display values.
• True and False values ->bool method
• sequence operations ->len andgetitem methods
• In Python, functions are first-class objects, so they can be passed around as data and have
attributes like any other object. Python also allows us to define objects that can be "called" like
functions by including acall method. With this method, we can define a class that behaves
like a higher-order function.
• Certain names are special because they have built-in behavior. These names always start and end
with two underscores.
init> Method invoked automatically when an object is constructed.
repr> Method invoked to display an object as a Python expression.
add> Method invoked to add one object to another.
bool> Method invoked to convert an object to true or False.
float> Method invoked to convert an object to a float / real number.
Special methods can also define the behavior of above built-in operators when they are applied to
user-defined objects.
2.7.3 Multiple Representations
• Object attributes, which are a form of message passing, allows different data types to respond
to the same message in different ways. A shared set of messages that elicit similar behavior
from different classes is a powerful method of abstraction. An interface is a set of shared
attribute names, along with a specification of their behavior.
• The requirement that two or more attribute values maintain a fixed relationship with each other
is a new problem. One solution is to store attribute values for only one representation and
compute the other representation whenever it is needed. Python has a simple feature for
computing attributes on the fly from zero-argument functions. The <b>@property</b> decorator allows

functions to be called without call expression syntax (parentheses following an expression).

@property

def magnitude(sel	f)	)	)
-------------------	----	---	---

• The interface approach to encoding multiple representations has appealing properties. The class for each representation can be developed separately; they must only agree on the names of the attributes they share, as well as many behavior conditions for those attributes. The interface is also additive: if another programmer wanted to add a third representation of complex numbers to the same program, they would have to create another class with the same attributes.

### 2.7.4 Generic Functions

- One way to implement cross-type operations is to select behavior based on the types of the
  arguments to a function or method. The idea of type dispatching is to write functions that
  inspect the type of arguments they receive.
- We use the type-tag attribute to distinguish types of arguments. One could directly use the built-in isinstance method as well, but tags simplify the implementation.
- Often the different data types are not completely independent, and there may be ways by which objects of one type may be viewed as being of another type. This process is called coercion. In general, we can implement this idea by designing coercion functions that transform an object of one type into an equivalent object of another type.
- Further advantages come from extending coercion. Some more sophisticated coercion schemes do not just try to coerce one type into another, but instead may try to coerce two different types each into a third common type. Another extension to coercion is iterative coercion, in which one data type is coerced into another via intermediate types. Chaining coercion in this way can reduce the total number of coercion functions that are required by a program.

## [3] Interpreting Computer Programs

## 3.1 Introduction

• A Python program is just a collection of text. Only through the process of interpretation do we perform any meaningful computation based on that text. A programming language like Python is useful because we can define an interpreter, a program that carries out Python's evaluation and execution procedures. It is no exaggeration to regard this as the most fundamental idea in programming, that an interpreter, which determines the meaning of expressions in a programming language, is just another program.

## 3.1.1 Programming Languages

- Powerful languages exist that do not include an object system, higher-order functions, assignment, or even control constructs such as while and for statements. An example is Scheme.
- Interpreters are programs that can carry out any possible computation, depending on their input. However, many interpreters have an elegant common structure: two mutually recursive functions, the first evaluate expressions in environments; the second applies functions to arguments. These functions are recursive in that they are defined in terms of each other: applying a function require evaluating the expressions in its body, while evaluating an expression may involve applying one or more functions.

# 3.2 Functional Programming

 Our object of study, a subset of the Scheme language, employs a very similar model of computation to Python's, but uses only expressions (no statements), specializes in symbolic computation, and employs only immutable values.

### 3.2.1 Expressions

Scheme programs consist of expressions, which are either call expressions or special forms. A call
expression consists of an operator expression followed by zero or more operand sub-expressions,
as in Python. Both the operator and operand are contained within parentheses.

(quotient 10 2)

5

- Scheme exclusively uses prefix notation. Operators are often symbols, such as + and \*. Call expressions can be nested, and they may span more than one line (space doesn't matter).
- The general form of an [if] expression is:

(if consequent> <alternative>)

To evaluate an if expression, the interpreter starts by evaluating the spredicate part of the expression. If the spredicate evaluates to a true value, the interpreter then evaluates the sconsequents and returns its value. Otherwise it evaluates the salternative and returns its values. (>= 2.1)

true

 The boolean values #t (or true) and #f (or false) in Scheme also can be connected by and, or, not.

### 3.2.2 Definitions

- Values can be named using the [define] special form. (define pi 3.14)
- New functions, called procedures in Scheme, can be defined using a second version of the define special form.

(define (square x)	(* x, x))													
The general form of a procedure definition is: (define ( <name> <formal parameters="">) <body>)</body></formal></name>														
(define ( <name> <formal parameters="">) <body>)</body></formal></name>														
• Anonymous functions are created using the lambda special form. Lambda is used to create														
procedure in the	e same v	vay as	define	e, ex	cept t	hat n	oʻnam	e is s	pecifi	ed fo	r the p	proced	lure.	
(lambda ( <formal-< td=""><td>paramet</td><td>ers&gt;) &lt;</td><td>body&gt;</td><td>) .</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></formal-<>	paramet	ers>) <	body>	) .										
3.2.3 Compound V	alues													
• Pairs are built	into the	Schem	e lang	guage	. Pair	s are	creat	ed wi	th the	e [con	s] buil	t-in f	unctio	n, and
the elements of	a pair d	are acc	essed	with	[car]	(for	the fi	rst el	emen	t) and	cdr (f	or the	e rest	(not
that the rest is														
element of the														
! cons plays a diff			ist: co	ons c	onnect	s elen	nent	1 to e	lemer	nt 2,	or addi	ng ele	ement	1 to
the front of element2; however, list creates a new list, with its first element pointing to element  1, second element pointing to element 2, etc (it can take infinite arguments).  2. Recursive lists are also built into the language using pairs. Every Scheme list is a linked list.														
• Recursive lists are also built into the language, using pairs. Every Scheme list is a linked list. A														
• Recursive lists are also built into the language, using pairs. Every Scheme list is a linked list. A special value denoted [nil] or ['()] represents the empty list. A recursive list value is rendered by														
placing its elem														
• Whether a list									fnull?	l pred	icate.		•	•
3.2.4 Symbolic Da							•					•	٠	•
• In order to man		symbol	s we i	need	a new	elem	ent in	our	anau	iae: t	he abi	litv ta	auote	e a
data object. ('a				1						.9			.7	
(define a 1)		(1		"		•		•			•	•	•	•
(define b 2)					•			•					•	
(define 5 b)					•		•						•	
(list a b) #we car	huild lie	sts wit	h the	built	· t_in lis	t pro	· ·edur	,						
(1 2)	i baila ti	319 1111	ii iiic	pan	i i ii iic	,, proc	-çuui (	•		•		•		•
(1 5)	•	•	•				•	•		•	•	•	٠	•
(list 'a 'b)			•	•	•		•	•		•		•	•	•
(a b)	•			٠	•						•		•	0
(a b)	•							•						
'(1 c) # same as (	'list 1 'c'		•	•				•			•	•	•	
(1 c) # same us (	(1131 1 0)	•	٠	•	•		•	•	•	•	•	٠	•	0
(list 1 c)			•				•	•					•	•
Error: unknown sy	mbol c			•							•	•	•	•
In Scheme, any ex		that	is not	oval	uatad	ie eai	1 to b		tad	•			•	0
• Quotation also									reu.	•	•			•
(car '(a b c))	allows us	io us	e com	μιιαι	10,115 1	النافاية	1 (11515	•		•				0
														•
(ada (a b a))													•	
(cdr '(a b c))				٠	0									
(b c)									•					
• We can also use	e comma	ro uno							•	•				
(define b 2)					٠							•	٠	
(a,b,c) #we used														
(a 2 c) #note tha	t this on	ly wor	KS IT V	we u	nquote	some	Thing	that	has a	n val	ue, oth	erwis	e Erro	r

'(a ,b c) #we used regular quote (a (unquote b) c) 3.2.5 Turtle Graphics

## 3.4 Interpreters for Language with Combination

## 3.4.1 A Scheme-Syntax Calculator

- The Calculator language is an expression language for the arithmetic operations of addition, subtraction, multiplication, and division.
- Subtraction (-) has two behaviors. With one argument, it negates the argument. With at least two arguments, it subtracts all but the first from the first. Division (/) has a similar pair of two behaviors: compute that multiplicative inverse of a single argument or divide all but the first into the first.

## 3.4.2 Expression Trees

- In order to write an interpreter, we must operate on expressions as data. A primitive expression is just a number or a string in Calculator: either an int or float or an operator symbol. A call expression is a Scheme list with a first element (the operator) followed by zero or more operand expressions.
- In Scheme, lists are nested pairs, but not all pairs are (well-formed) lists. A well-formed list is a pair whose second element is either a list or nil. We notate a pair that is not a well-formed list by adding a dot between any two of its elements. When pairs represent well-formed lists, they have the method [len] (in other words, if len is applied to pairs, it will cause Error).
- The Calculator language has primitive expressions and call expressions. A primitive expression is a
  number; a call expression is a combination that begins with an operator followed by one more
  expressions. Expressions are represented as Scheme lists that encode tree structures.
- The value of a Calculator expression is defined recursively. A primitive expression (number) is self-evaluating. And a call expression evaluates to its argument values combined by an operator.

## 3.4.3 Parsing Expressions

- Parsing is the process of generating expression trees from raw text. A parser is a composition of
  two components: a lexical analyzer and a syntactic analyzer. First, the lexical analyzer partitions
  the input string into tokens (white space is ignored), which are the minimal units of the language,
  such as names and symbols. Second, the syntactic analyzer constructs an expression tree from
  this sequence of tokens. Lexical analysis is an iterative process, and syntactic analysis is a treerecursive process.
- A predictive recursive descent parser inspects only k tokens to decide how to proceed, for some
  fixed k. The Scheme language is exactly something can be parsed by it. Syntactic analysis
  identifies the hierarchical structure of an expression, which may be nested. Each call to
  scheme\_read consumes the input tokens for exactly one expression. The base case is symbols and
  numbers. The recursive call is to call scheme\_read on sub-expressions and combine them.

### 3.4.4 Calculator Evaluation

- The eval function computes the value of an expression, which is always a number. It is a generic
  function that dispatches on the type of the expression (primitive or call).
- The apply function applies some operation to a (Scheme) list of argument values. In Calculator, all operations are named by built-in operators: +, -, \*, /.
- The user interface for many programming languages is an interactive interpreter.

- 1. Print a prompt
- 2. Read text input from the user
- 3. Parse the text input into an expression
- 4. Evaluate the expression
- 5. If any errors occur, report those errors, otherwise
- 6. Print the value of the expression and repeat
- Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply.
- 1. Lexical analysis: the token 2.3.4 raises ValueError.
- 2. Syntactic analysis: an extra ) raises SyntaxError.
- 3. Eval: an empty combination () raises TypeError.
- 4. Apply: no arguments to raises TypeError.
- A well-formed interactive interpreter should not halt completely on an error, so that the user
  has an opportunity to try again in the current environment.
- 3.5 Interpreters for Languages with Abstraction
- 3.5.1 Structure
- Eval:

#### Base cases:

- 1. Primitive values (numbers)
- 2. Look up values bound to symbols

#### Recursive calls:

- 1. Eval (operator, operands) of call expressions
- 2. apply (procedure, arguments)
- 3. eval (sub-expressions) of special forms
- Apply:

### Base cases:

1. Built-in primitive procedures

# recursive calls:

- 2. Eval (body) of user-defined procedures
- The scheme\_eval function dispatches on expression form:
- 1. Symbols are bound to values in the current environment.
- 2. Self-evaluating expressions are returned.
- 3. All other legal expressions are represented as Scheme lists, called combinations. (Examples: if, lambda, define: special forms are identified by the first list element.)
- 3.5.2 Environments
- The above structure of an interpreter requires an environment for symbol lookup. It creates a new environment each time a user-defined procedure is applied. And that environment will be passed into eval as an argument so that when we look up a name, we get the right value for it.
- A frame represents an environment by having a parent frame. Frames are Python instances methods lookup and define. In Scheme, frames do not hold return values. The frame without any parent is the global frame.
- Define binds a symbol to a value in the first frame of the current environment.

### (define <name> <expression>)

- 1. Evaluate the <expression>
- 2. Bind <name> to its value in the current frame

3.!	argum	ent vonvironment of the tareast tareas	alues, ment i proce Progi ns may pecial	whose in whicedure rams y only form	e pare ch the in the evalu	ent is e proc e envir uate s uates	the e edure ronme ome s to the	nv (an was nt the ub-ex e quot	origin origin at sta press ed ex	bute ally do rts wi ions, li pressi	of the efined ith th ike if, ion, w	e proc l) of t is new and,	edure he pro fram  or, co	objectocedurine.		rem	ember	
•	•		•													۰	٠	•
0	0			•		•	٠	0				٠	•	•		٠	0	
•	0						۰	0								0	0	
	0			•			٠	٠	٠					•		٠	٠	
•	0			٠		٠	٠	٠	٠	٠		٠	٠		,	•	٠	
	•					٠	•	•				•					•	
										•								
					•													
•	0						٠	0	0	٠							0	
0	a		۰			٠	0			0	٠	٠					0	0
																		•
															۰			
•	٠	•	•	•	•	•	•	•	٠	•		•	•	•	•	•	•	
			•	•		•	•			•			•				٠	
	•			•	•		•	٠		•				•			•	
			٠	•		٠	٠	٠		٠			•				٠	
٠							•	0	0	٠	•				•		0	
0							٠	•								0	6	