Anthropic Claude SDK

The Anthropic Python SDK at a glance, from a single Claude message to streaming, tool use, images, and prompt caching.

Client & First Message

Construct the client, send one turn, read the text back.

PURPOSE	COMMAND	RESULT
Construct the client (reads env key).	`client = anthropic.Anthropic()`	
Send the simplest message.	`msg = client.messages.create(model=..., max_tokens=1024, messages=[{...}])`	
Read the first text block.	`msg.content[0].text`	
Walk all text blocks safely.	`for b in msg.content:` `if b.type == "text": print(b.text)`	
See why it stopped.	`msg.stop_reason`	
Check token usage.	`msg.usage.input_tokens, .output_tokens`	

client.messages.create(...) returns a Message whose .content is a list of typed blocks; read .stop_reason and .usage for why and what it cost

Messages & Roles

The conversation is a list of role-tagged turns you resend every call.

PURPOSE	COMMAND	RESULT
One user turn (shorthand).	`messages = [{"role": "user",` `"content": "..."}]`	
Multi-turn history (alternating).	`messages = [{"role":"user", ...},` `{"role":"assistant", ...}, ...]`	
Append the model's reply back.	`messages.append({"role":"assistant", ...})`	
Add the next user turn.	`messages.append({"role":"user", ...})`	
Content can be a block list.	`"content": [{"type":"text", "text":"Hi"}]`	
First turn must be user.	`messages[0]["role"] == "user"`	

The API is stateless: a conversation is a list of {role, content} turns, always starting with user, that you resend in full every call

System Prompt & Parameters

Steer behavior with system, bound output with max_tokens, deepen reasoning with thinking.

PURPOSE	COMMAND	RESULT
Set a system prompt.	`create(..., system="You are terse.")`	
Bound the output length.	`create(..., max_tokens=1024)`	
Turn on adaptive thinking.	`thinking={"type": "adaptive"}`	
Tune effort vs cost.	`output_config={"effort": "high"}`	
Show summarized reasoning.	`thinking={"type": "adaptive", "display": "summarized"}`	
Pick the model.	`model="claude-opus-4-8"`	

system steers, max_tokens is a required hard cap; thinking={"type": "adaptive"} deepens reasoning (hidden unless display="summarized")

Streaming

Stream tokens as they arrive; collect the final Message at the end.

PURPOSE	COMMAND	RESULT
Open a streaming context.	`with client.messages.stream(model=..., max_tokens=1024, messages=msgs) as stream:`	
Print text as it arrives.	`for text in stream.text_stream: print(text, end="", flush=True)`	
Get the complete Message at the end.	`final = stream.get_final_message()`	
Handle event types by hand.	`for event in stream: if event.type== "content_block_delta": ...`	
Why stream at all.	`# long outputs / high max_tokens`	
Stream with thinking visible. `thinking display="summarized"`	`thinking={"type": "adaptive", "display": "summarized"}; stream.text_stream`	

with client.messages.stream(...) as stream: iterate stream.text_stream for tokens, then get_final_message() for the whole Message

Tool Use

Define tools, get a tool_use block, run it, send a tool_result back.

PURPOSE	COMMAND	RESULT
Define a tool (JSON schema).	`tools = [{"name": "get_weather", "input_schema": { "city": str }}]`	
Offer the tools on a call.	`messages.create(..., tools=tools)`	
Model asks to call a tool.	`msg.stop_reason == "tool_use"`	
Run it and build the result.	`{"type": "tool_result", "tool_use_id" : block.id, "content": ...}`	
Send the result, loop until done.	`messages += [assistant,tool_result]` `create(...) again until end_turn`	
Let the SDK run the loop.	`client.beta.messages.tool_runner(...)`	

tools= sends JSON schemas; stop_reason "tool_use" carries a tool_use block; reply with a matching tool_result and loop to end_turn

Multimodal (Images)

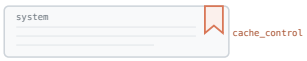
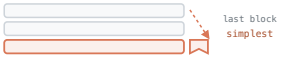
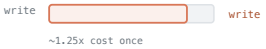
Put image blocks in the content list, base64 or URL, alongside text.

PURPOSE	COMMAND	RESULT
Encode a local image.	`data = base64.standard_b64encode(` `open("cat.png","rb").read()).decode()`	
Send a base64 image block.	`{"type": "image", "source": {"type": "base64", "media_type":"image/png", "data":data}}`	
Send an image by URL.	`{"type": "image", "source": {"type": "url", "url":"https://.../cat.png"}}`	
Mix image and a question.	`content=[image_block, {"type": "text", ...}]`	
Read the description back.	`msg.content[0].text`	
Supported formats note.	`# jpeg, png, gif, webp`	

An image is just another content block in a user turn: base64 or url, sat beside your text block, answered as ordinary text

Prompt Caching

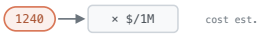
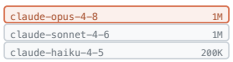
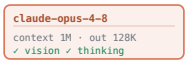
Mark a stable prefix with cache_control to reuse it cheaply across calls.

PURPOSE	COMMAND	RESULT
Cache a big system prefix.	`system=[{"type":"text","text":BIG_DOC, "cache_control":{"type":"ephemeral"}}]`	
Let the SDK auto-place it.	`client.messages.create(..., cache_control={"type":"ephemeral"})`	
First call writes the cache.	`msg.usage.cache_creation_input_tokens`	
Later calls read the cache.	`msg.usage.cache_read_input_tokens`	
Pick a longer TTL.	`"cache_control":{"type":"ephemeral", "ttl":"1h"}`	
Cache is a strict prefix match.	`# any byte change before the mark = miss`	

cache_control marks a stable prefix: the first call writes it (~1.25x), later identical-prefix calls read it (~0.1x); any byte change before the mark misses

Token Counting & Models

Count before you send; pick the model by cost, context, and capability.

PURPOSE	COMMAND	RESULT
Count tokens before sending.	`client.messages.count_tokens(messages=…)`	
Read the count.	`count.input_tokens`	
List available models.	`client.models.list()`	
Inspect one model's limits.	`client.models.retrieve(id).max_input_tokens`	
Choose by tier.	`model= opus vs sonnet vs haiku`	
Avoid deprecated spellings.	`budget_tokens= · temperature= thinking={"type": "adaptive"}`	

count_tokens sizes a prompt cheaply; models.list / retrieve read live limits; pick opus / sonnet / haiku by capability, context, and cost